NPS-SW-00-001

NAVAL POSTGRADUATE SCHOOL
Monterey, California

System Engineering and Evolution Decision Support

Interim Progress Report (01/01/2000 – 09/30/2000)

By

Luqi

September 2000

Prepared for: U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM David R. Ellison
Superintendent

Richard S. Elster
Provost

This report was prepared for U.S. Army Research Office and funded in part by the U.S. Army Research Office.

Prepared by:

Luqi
Professor, Computer Science

Reviewed by:

Luqi
Director, Software Engineering
Automation Center

Reviewed by:

Dan Boger
Dean of Computer and Information Sciences
and Operations

Released by:

D. W. Netzer
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 09/30/2000	3. REPORT TYPE AND DATES COVERED Interim Progress Report 01/01/2000 – 09/30/2000

4. TITLE AND SUBTITLE

System Engineering and Evolution Decision Support –
Interim Progress Report (01/01/2000 – 09/30/2000)

5. FUNDING NUMBERS

38690-MA

6. AUTHOR(S)

Professor Luqi

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Automation Center,
Naval Postgraduate School, Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

NPS-SW-00-001

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U. S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of our effort is to develop a scientific basis for system engineering automation and decision support. This objective addresses the long term goals of increasing the quality of service provided complex systems while reducing development risks, costs, and time. Our work focused on decision support for designing operations of complex modular systems that can include embedded software. Emphasis areas included engineering automation capabilities in the areas of design modifications, design records, reuse, and automatic generation of design representations such as real–time schedules and software.

14. SUBJECT TERMS

System Engineering, Decision Support, Evolution, Concurrent Engineering

15. NUMBER OF PAGES

159

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION ON THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Table of Contents

Interim Progress Report

System Engineering and Evolution Decision Support

1/1/2000 - 9/30/2000

Luqi

List of Manuscripts:

V. Berzins, M. Shing, Luqi, M. Saluto, and J. Williams, "Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping", in *Design Automation for Embedded Systems*, 5(3/4), August 2000, pp.251-263.

V. Berzins, M. Shing, Luqi, M. Saluto, and J. Williams, "Object-oriented modular architecture for ground combat simulation", in *Proceedings of the 2000 Command and Control Research and Technology Symposium*, Naval Postgraduate School, Monterey, CA, June 26-28, 2000.

V. Berzins, "Static Analysis for Program Generation Templates", in *Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/*

J. Gou and Luqi, "Reuse and Re-engineering of Legacy Systems", in *Proceedings of the 5th World Conference on Integrated Design & Process Technology*, Dallas, TX, June 4-8,2000.

J. Guo, and Luqi, "A Survey of Software Reuse Repositories", in *Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS'2000)*, Edinburgh, Scotland, UK, April 6-7, 2000.

Luqi and J.C. Nogueira, "A Risk Assessment Model for Evolutionary Software Projects", in Proceedings of 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000. Also available on-line at http://www.disi.unige.it/person/ReggioG/PROCEEDINGS/

Luqi, V. Berzins, M. Shing, R. Riehle, and J.C. Nogueira, "Evolutionary Computer Aided Prototyping System (CAPS)", in *Proceedings of the TOOLS USA 2000 Conference*, Santa Barbara, CA, July 30-August 3, 2000.

Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo, and B. Shultes, "The Use of Computer Aided Prototyping for Re-engineering Legacy Software", submitted to the *IEEE Transaction on Software Engineering*.

N. Nada, Luqi, D. Rine, and K. Jaber, "Product Line Stakeholder Viewpoint and Validation Models", in *Proceedings of the Workshop on Software Product Lines: Economics, Architectures, and Implications*, Limerick, Ireland, June 4-11, 2000.

N. Nada, Luqi, D. Rine, and E. Damiani, "A Knowledge-Based System for Software Reuse Technology Practices", in *Proceedings of the Third International Workshop on Intelligent Software Engineering (WISE3)*, Limerick, Ireland, June 4-11, 2000.

J.C. Nogueira, Luqi, and V. Berzins, "Risk Assessment in Software Requirement Engineering", in *Proceedings of the 5th World Conference on Integrated Design & Process Technology*, Dallas, TX, June 4-8,2000.

J.C. Nogueira, C. Jones, and Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering", in *Proceedings of the 2000 Command and Control Research and Technology Symposium*, Monterey, CA, June 26-28, 2000.

J.C. Nogueira, Luqi, V. Berzins, and N. Nada, "A formal Risk Assessment Model for Software Evolution", in *Proceedings of the 2nd International Workshop on Economics-Driven Software Engineering Research (EDSER-2)*, Limerick, Ireland, June 4-11, 2000.

J.C. Nogueira, Luqi, and S. Bhattacharya, "A Risk Assessment Model for Software Prototyping Projects", in *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping*, Paris, France, June 21-23, 2000.

Scientific Personnel:

Dr. Du Zhang, Visiting Professor, NPS.

Dr. Swapan Bhattacharya (National Research Council Research Associate)

Dr. Jiang Guo (National Research Council Research Associate)

Dr. Jun Ge (National Research Council Research Associate)

Dr. Mikhail Auguston (National Research Council Research Associate)

Dr. Oleg Kiselyov (National Research Council Research Associate)

J.C. Nogueira, "A Formal Model for Risk Assessment in Software Projects", Doctoral Dissertation, Software Engineering, NPS, September 2000.

Report of Inventions: N/A

Scientific Progress and Accomplishments:

The objective of our research is to develop an integrated set of formal models and methods for system engineering automation. These results will enable building decision support tools for concurrent engineering. Our research addresses complex modular systems with embedded control software and real-time requirements.

We focused on automation of design activities that appear in an evolutionary approach to system development. Decision support for design synthesis, reuse and evolution is emphasized. This research extended recently developed formal methods in system engineering to construct a cohesive set of formal models. These models are used to create and to connect automated processes for computer aided prototyping, requirements validation, and design synthesis. Mathematical models for implementing a set of automated and integrated engineering automation tools were also developed. Our work combined very-high-level specification abstractions and concepts with: (1) formal real-time models, (2) automated management of system design data and human resources, (3) design transformations, (4) change merging, (5) automated retrieval of reusable system design components, and (6) automated schedule construction. We have created automated methods for: (1) generating real-time control programs, (2) generating simulations of subsystems, and (3) coordinating concurrent work by engineering teams. Our work will ensure design consistency and to alleviate communication difficulties.

The significance of our work is to:

- improve system effectiveness and flexibility,
- increase engineering productivity, and
- reduce system maintenance costs.

This was achieved by providing a higher level of engineering automation coupled directly with requirements validation facilities. Our work will broaden the scope of engineering decision support to include concurrent whole-system engineering, requirement determination, and system evolution. Automated decision support will ensure system quality by decreasing the human effort required. This, in turn, will minimize the incidence of human error. The trial use of operational system prototypes linked with software simulations of selected subsystems enables users to provide feedback for validation and refinement of system requirements prior to detailed design. Maintenance costs can be minimized by reducing the need to repair requirement errors after system deployment. We provided methods for process and system re-engineering at minimal cost. This was achieved by: (1) regenerating new variations of designs from high-level decisions. (2) combining changes, and (3) propagating the consequences of design modifications. These engineering capabilities will enable the Army to improve and integrate its complex systems with reduced costs. Improved systems can reduce Army manpower needs while strengthening information warfare capabilities.

Specific Tasks accomplished in FY00 include (1) the development of a risk assessment model for the evolutionary software process; (2) a detailed survey of the software reuse repositories, (3) the development of models to support reuse in product line approach, and (4) tool enhancements for system engineering and evolution decision support.

Technology Transfer:

V. Berzins, member, Steering Committee, 2000 ARO/NSF/CNR Monterey Workshop On Modeling Software System Structures in a Fastly Moving Scenario, held in Santa Margherita Ligure, Italy, June 13-16, 2000.

V. Berzins, "Static Analysis for Program Generation Templates", presented at the *7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000.*

V. Berzins, "A formal Risk Assessment Model for Software Evolution", in presented at the 2nd International Workshop on Economics-Driven Software Engineering Research (EDSER-2), Limerick, Ireland, June 4-11, 2000.

J. Gou, "Reuse and Re-engineering of Legacy Systems", presented at the *5th World Conference on Integrated Design & Process Technology*, Dallas, TX, June 4-8,2000.

Luqi, "A Survey of Software Reuse Repositories", presented at the *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS'2000)*, Edinburgh, Scotland, UK, April 6-7, 2000.

Luqi, "A Risk Assessment Model for Evolutionary Software Projects", presented at the 7th Monterey Workshop "Modeling Software System Structures in a fastly moving scenario", Santa Margherita Ligure, Italy, June 13-16, 2000.

Luqi, "A Risk Assessment Model for Software Prototyping Projects", presented at the 11th IEEE International Workshop on Rapid System Prototyping, Paris, France, June 21-23, 2000.

Luqi, Chair of the Program Committee, the 11th IEEE International Workshop on Rapid System Prototyping, held in Paris, France, June 21-23, 2000.

Luqi, Co-Chair, Program Committee, 2000 ARO/NSF/CNR Monterey Workshop On Modeling Software System Structures in a Fastly Moving Scenario, held in Santa Margherita Ligure, Italy, June 13-16, 2000.

J.C. Nogueira, "Risk Assessment in Software Requirement Engineering", presented at the 5th World Conference on Integrated Design & Process Technology, Dallas, TX, June 4-8,2000.

J.C. Nogueira, "Surfing the Edge of Chaos: Applications to Software Engineering", presented at the 2000 Command and Control Research and Technology Symposium, Monterey, CA, June 26-28, 2000.

R. Riehle, "Evolutionary Computer Aided Prototyping System (CAPS)", presented at the TOOLS USA 2000 Conference, Santa Barbara, CA, July 30-August 3, 2000.

R. Riehle, chair of the Tutorial Committee, the TOOLS USA 2000 Conference, Santa Barbara, CA, July 30-August 3, 2000.

M. Shing, "Object-oriented modular architecture for ground combat simulation", presented at the *2000 Command and Control Research and Technology Symposium*, Naval Postgraduate School, Monterey, CA, June 26-28, 2000.

M. Shing, member of the Program Committee, the 11th IEEE International Workshop on Rapid System Prototyping, held in Paris, France, June 21-23, 2000.

APPENDICES

Design Automation for Embedded Systems, 5, 251–263 (2000)
© 2000 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping

VALDIS BERZINS berzins@cs.nps.navy.mil
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943

MAN-TAK SHING mantak@cs.nps.navy.mil
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943

LUQI luqi@cs.nps.navy.mil
Computer Science Department, Naval Postgraduate School, Monterey, CA 93943

MICHAEL SALUTO dm5447@exmail.usma.army.mil
EECS Department, United States Military Academy, West Point, NY 10996

JULIAN WILLIAMS julian_williams@jsims.mil
JSIMS Joint Program Office, 12249 Science Dr., Suite 260, Orlando, Fl 32826

Abstract. This paper describes a case study to determine whether computer-aided prototyping techniques provide a cost-effective means for re-engineering legacy software. The case study consists of developing an object-oriented modular architecture for the existing US Army Janus combat simulation system, and validating the architecture via an executable prototype using the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School. The case study showed that prototyping can be a valuable aid in the re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed. The CAPS system enabled us to do this with a minimal amount of coding effort.

Keywords: Computer-aided prototyping, software re-engineering, software evolution, object-oriented architecture, combat simulation.

1. Introduction

This paper describes a case study to determine whether computer-aided prototyping techniques provide a cost-effective means for re-engineering legacy software [14]. The case study consists of developing an object-oriented modular architecture for the existing Janus(A) system [6], and validating the architecture via an executable prototype using the Computer Aided Prototype System (CAPS) [10, 11].

Janus(A) is a software-based war game that simulates ground battles between up to six adversaries. It is an interactive, closed, stochastic, ground combat simulation that features precise color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. It has gone through six major revisions since 1978. The current version of Janus operates on Hewlett Packard workstations and consists of a large number of FORTRAN modules (1918 FORTRAN routines, 115 C routines, and a total of 393K lines of source code), organized as a flat structure and interconnected with one another via 129 FORTRAN

COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. There is a need to modernize the Janus software into a maintainable and evolvable system (written in C++) and to take advantage of modern Personal Computers to make Janus more accessible to the Army. The TRACDOC Analysis Center (TRAC) initiated the HLA Warrior project in 1998 to re-engineer Janus into an HLA compliant, PC-based combat simulation, with improved graphical user interface, object-oriented source code, and a modern modular architecture [13]. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to produce an object-oriented architecture that supports existing and required enhancements to Janus functionality. The architecture provides protocols for communication between the graphical user interface and the simulation models and acts as a blueprint for developing the C++ code.

The paper is organized as follows. We present the re-engineering process and the resultant object-oriented architecture in Sections 2 and 3. Section 4 describes our prototyping experiment. Section 5 summarizes the lessons learned and Section 6 draws some conclusions.

2. The Re-engineering Process

Software re-engineering is the process of creating an abstract description of a system, reasoning about a change at a higher level of abstraction, and then re-implementing the system [5]. This section describes the first two activities of the re-engineering process.

2.1. *Reverse-Engineering*

The first step in reverse-engineering is system understanding, which was accomplilshed via a series of brief meetings with the client, TRAC-Monterey. We asked questions and made notes on the system's operation and its current functionality. We paid particular attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. Additionally we collected copies of the Janus User's manual, the Janus Programmer's Manual, the Janus Database Management Program Manual, the Janus Software Design Manual, and the Janus Algorighm Document [6–9, 12].

The next step is to abstract the system's functionality and then produce system models that accurately represent that functionality. Analysis of 393K lines of legacy code is a daunting but inescapable part of the process. We recoiled from the magnitude of this effort in the beginning of the project and relied on information contained in the Janus manuals. In hindsight, it was a mistake that slipped the schedule of the project by serveral months. While these documents helped us get started because they contained higher level information and were much shorter than the code, they were much older and contained outdated information. We should have started analyzing the source code right away and should have persistently continued with this task in parallel with all other re-engineering activities.

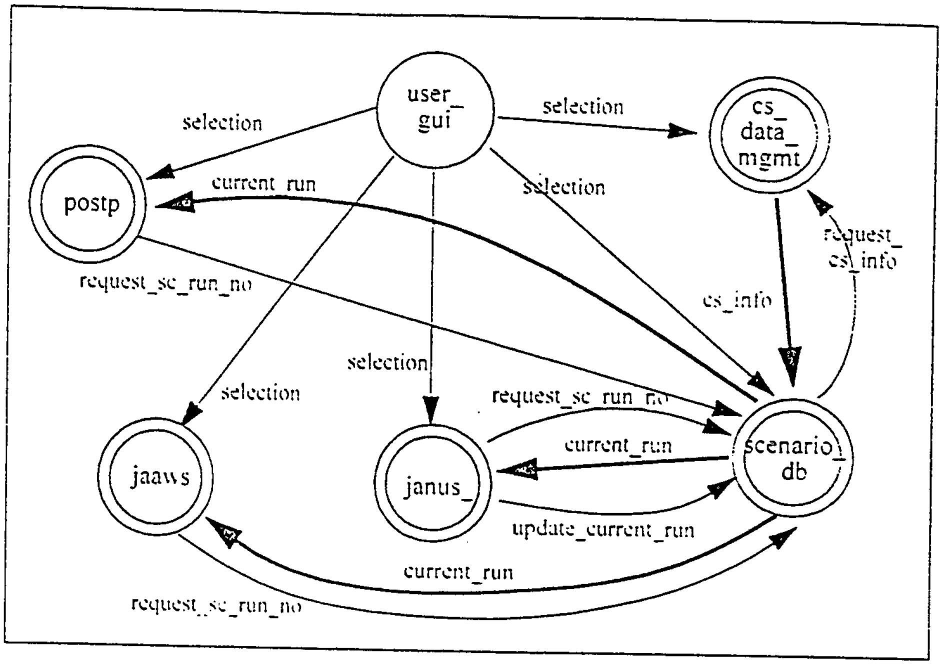


Figure 1. Top-level communication structure of the existing Janus software.

We divided the Janus source code by directories amongst the team members to explore, examine and gather information. Using strictly manual techniques and review procedures, we were able to get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [7] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs. Using that knowledge, we developed functional models from the data flows.

We used the Computer-Aided Prototyping System (CAPS), an automated tool developed at the Naval Postgraduate School, to assist in developing the abstract models. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to develop our diagrams separately, and then join them together under the CAPS environment, where they can be used to generate an executable model of the architecture. Figure 1 shows the resultant top-level structure of the existing Janus system. It consists of five subsystems—*cs_data_mgmt*, *scenario_db*, *janus*, *jaaws*, and *postp*. The *cs_data_mgmt* subsystem manages combat system databases. The *scenario_db* subsystem manages the different scenarios and simulation runs in the system. The *janus* subsystem simulates the ground battles. The *jaaws* subsystem allows analysts to perform post-simulation analysis and the *postp* subsystem allows Janus users to view simulation reports.

2.2. *Transformation of Functional Models to Object Models*

Next, we developed object models of the Janus system, using the aforementioned materials and products to create the modules and associations amongst them. This was probably the most difficult and most important step. It required a great deal of analysis and focus to transform the originally scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. This was a crucial step because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software. We first identified a set of candidate objects and created an object model for the core elements based on the information from the Database Management Program Manual [8] and the domain knowledge of the human experts. Then we analyzed the source code and used the information from the Software Design Manual [9] to add attributes and operations to the object classes. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and/or supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

2.3. *Refinement of the Object Models and the Development of the Object Oriented Architecture*

During this phase of the project, the re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core elements and the object-oriented architecture for the Janus system. They presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center. Many researchers have reported that domain knowledge plays a critical role during the software re-engineering process [2–4]. Since we were not familiar with the domain of ground combat simulation, we found that these meetings were invaluable to our project. Our experience supports the ideas that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation [1], because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort. Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus system (Figure 2).

3. Software Architecture for the Janus Combat Simulation System

Central to the existing Janus Combat Simulation subsystem is the program RUNJAN, which is the main event scheduler for the Janus simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event,

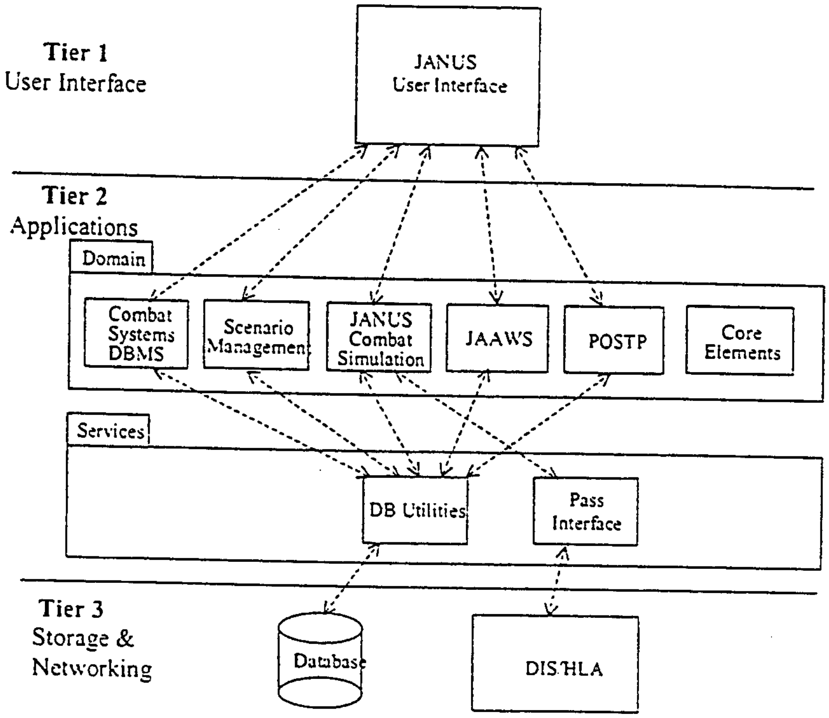


Figure 2. The proposed 3-tier object-oriented architecture.

RUNJAN advances the game clock to the scheduled time of the event and performs that event. The existing event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation subsystem is to distribute the event handling functions to individual objects. Moreover, it is necessary to redefine some event categories to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Interactions between the simulation engine and the world modeler (the interface to a distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation.

The new architecture uses an explicit priority queue of event objects to schedule the simulation events. Each event object has an associated simulation object, which is the target of the event. There are 14 event groups, which correspond to the 14 event subclasses shown in Figure 3.

An object-oriented approach enabled us to reduce the number of event types needed in the simulation. Depending on the subclass which an event object belongs to, the Execute method will invoke the corresponding event handler of the associated simulation object to handle

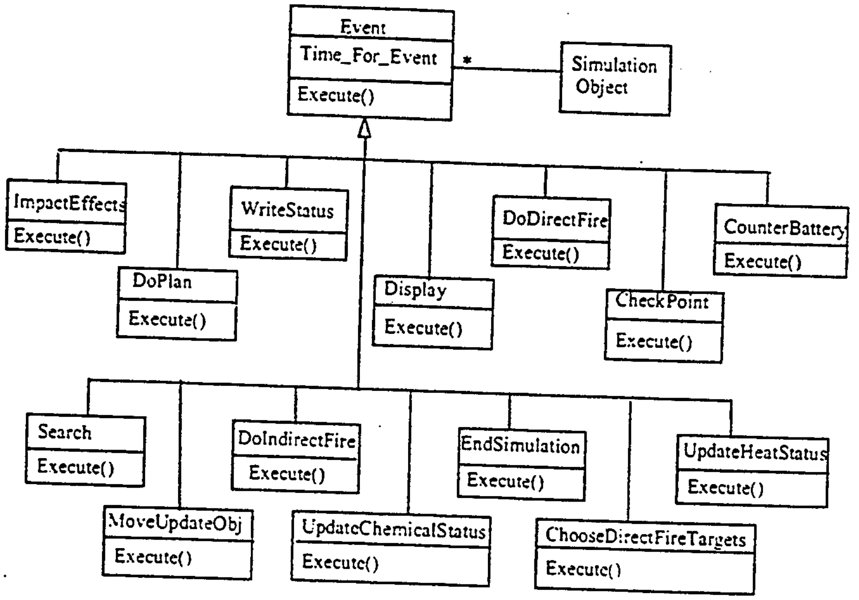


Figure 3. The event class hierarchy.

the event (Figure 4). The simulation object superclass defines the interface of the event handlers for the event groups, and provides an empty body as the default implementation for the event handlers. The methods are overridden by the actual event handler code at the subclasses that have non-empty actions associated with the events.The above architecture enables a very simple realization of the main simulation loop:

```
initialization;
while not_empty(event_queue) loop
      e := remove_event(even_queue);
      e.execute( );
end loop;
finalization;
```

Note that this same code handles all kinds of events, including those for future extensions that are yet to be designed. Event objects are created and inserted into the event queue by the initialization procedure at the beginning of the simulation, by the constructors of new simulation objects, and by the actions of other event handlers. Depending on the actual implementation of when and how events are inserted into the priority event queue, it may be necessary to allow events to change their priorities while waiting in the queue.

World Model object subclasses (with names starting with the "WM" prefix) are created to provide specialized methods for the world modeler to update the objects from other simulators. Information concerning objects local to the Janus simulator can be broadcast

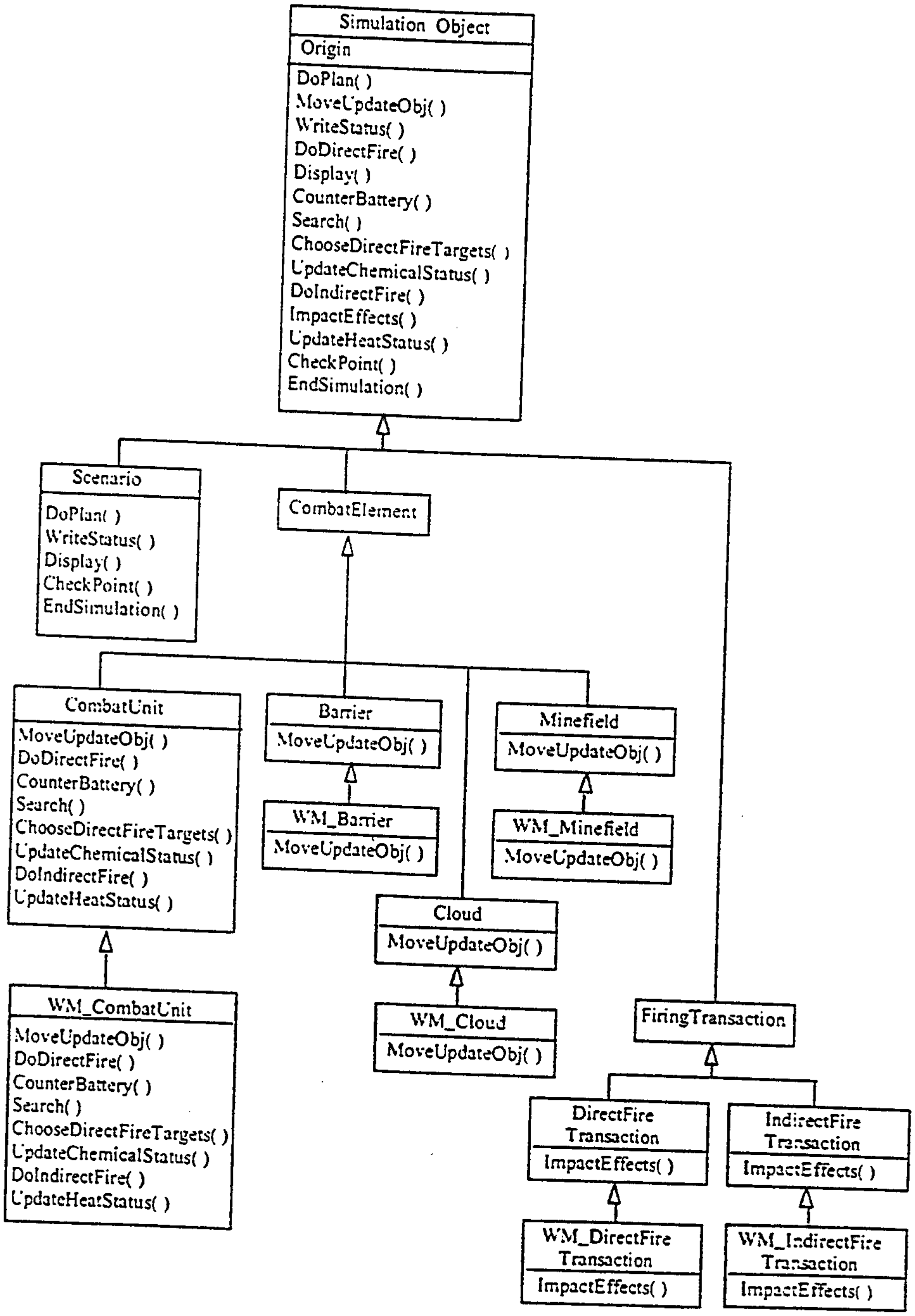


Figure 4. The simulation object class hierarchy.

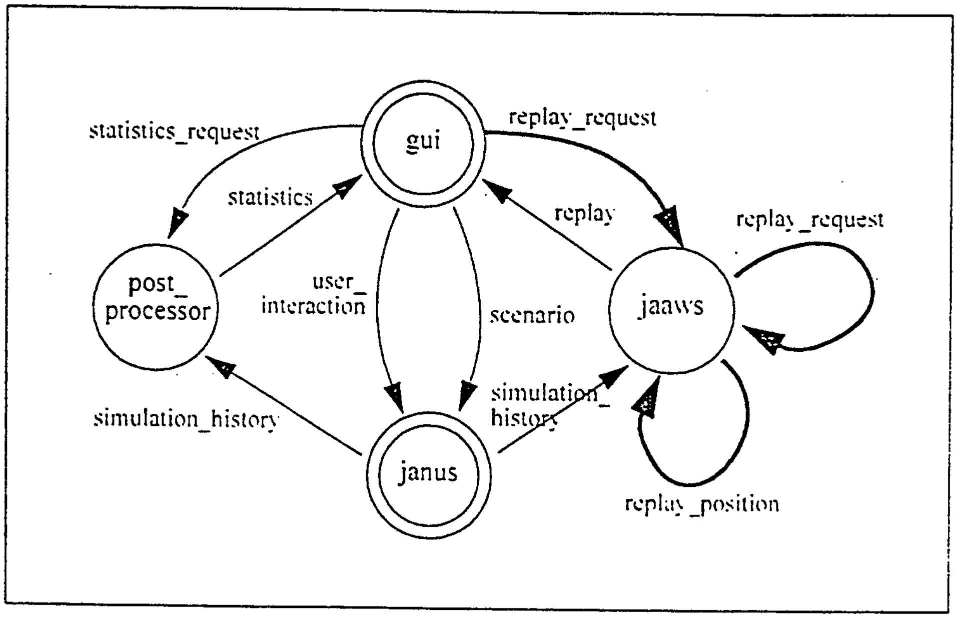


Figure 5. Top-level decomposition of the executable prototype.

over the simulation network either periodically by an active world modeler object, or by individual local objects whenever they update their own states.

4. Development of an Executable Prototype Using CAPS

In order to validate the proposed architecture and to refine the interfaces of the Janus subsystems, we developed an executable prototype using CAPS. Figure 5 shows the top-level structure of the prototype, which has four subsystems: Janus, GUI, JAAWS and the POST-PROCESSOR. Among these four subsystems, the Janus and the GUI subsystems (depicted as double circles) are made up of sub-modules shown in Figures 6 and 7, while the JAAWS and the POST-PROCESSOR subsystems (depicted as single circles) are mapped directly to objects in the target language. After entering the prototype design using CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems.

Due to time and resource limitations, we only developed the prototype for a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (MoveUpdateObj, DoPlan, and EndSimulation), and one kind of post-processing statistics (fuel consumption). In addition, a simple user interface was developed using TAE [15] (Figure 8). The resultant prototype has over 6000 lines of program source code and contains enough features to exercise all parts of the architecture.

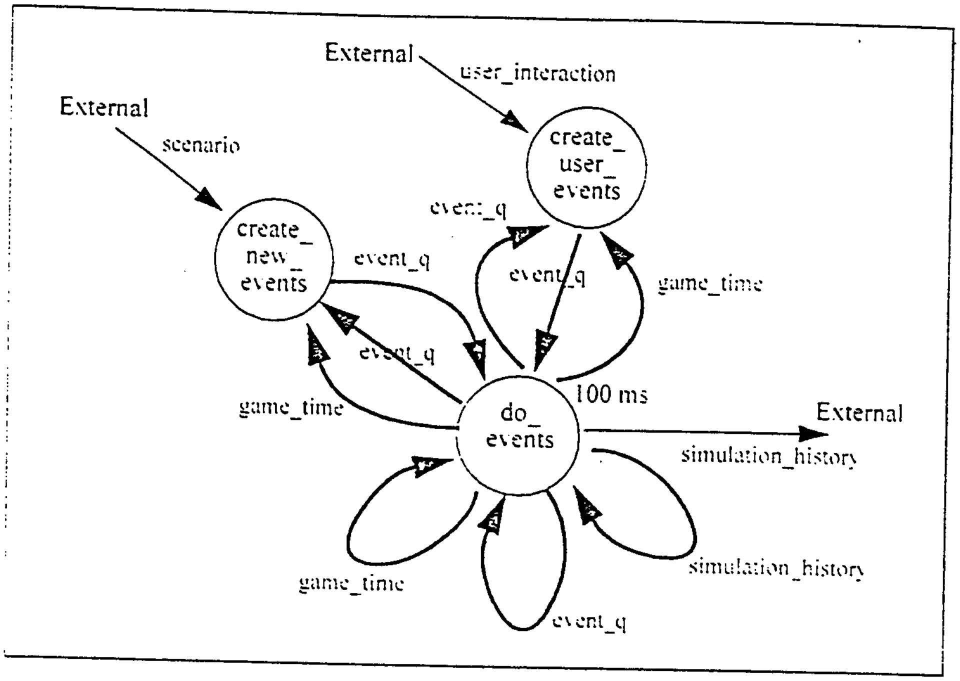


Figure 6. The JANUS subsystem of the executable prototype.

The code that handles the motion of a generic simulation object was very simple, but it was designed so that it would work in both two and three dimensions without modification (currently the initialization and the movement plan of the tank object never call for any vertical motion). The code was also designed to be polymorphic, just as was the main event loop. This means the same code will handle the motion of all kinds of simulation objects without any modifications, including even new types of simulation objects that are part of future enhancements to Janus and have not yet been designed or implemented.

5. Lessons Learned

Our prototyping experiment showed that the proposed object-oriented architecture allows design issues to be localized and provides easy means for future extensions. We started out with a prototype consisting of only two event subclasses (MoveUpdateObj and End-Simulation) and were able to add a third event subclass (DoPlan) to the prototype without modifying the event control loop of the Janus combat simulator.

We also demonstrated the use of inheritance and polymorphism to efficiently extend/specialize the behavior of combat units. For example, to implement the MoveUpdateObj method of a tank subclass which uses the general-purpose method from its superclass to compute its distance traveled and a specialized algorithm to compute its fuel consumption,

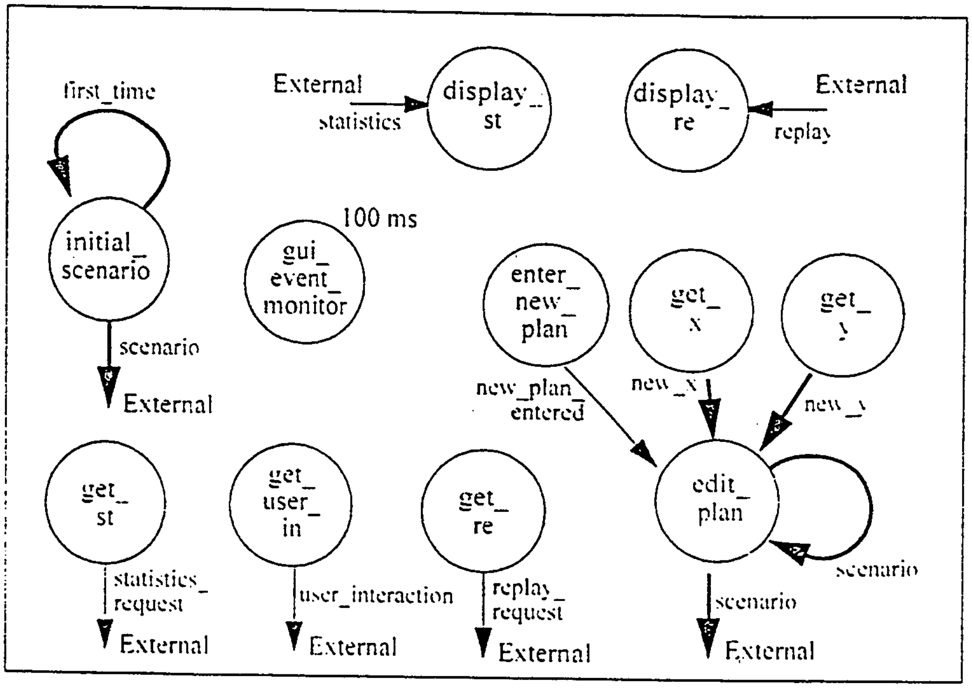
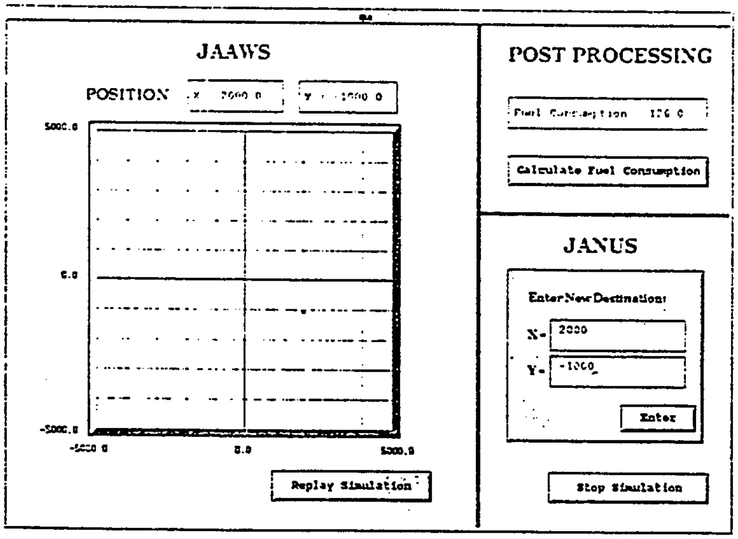


Figure 7. The GUI subsystem of the executable prototype.


JAAWS

POSITION X 2000.0 Y -1000.0

POST PROCESSING

Fuel Consumption 176.0

JANUS

Enter New Destination:

X= 2000

Y= -1000

Figure 8. The Graphical User Interface of the executable prototype.

we simply include 1 statement to invoke the MoveUpdateObj method of its superclass followed by three lines of code to update its fuel consumption. Moreover, other combat unit subclasses can be added easily to the prototype without the need to modify the event scheduling/dispatching code.

The prototype also resulted in the following refinements to the proposed architecture:

(1) Instead of a procedure with no return value, change the Execute operation to return the time at which the next event is to be scheduled for the same simulation object, and introduce a special time value "NEVER" to indicate that no next event is needed. The proposed change turns the communication between the event dispatcher and the simulation objects from a peer-to-peer communication into a client-server communication. This change eliminates the dependency of simulation objects on details of the event queue and allows the event dispatcher to use a single statement to schedule all recurring events for all event types.

(2) Instead of recording the history of a simulation run in terms of sets of data files, model the simulation history as a sequence of events. The proposed change provides a simple and uniform way to handle history records for all events, and allows the same modular architecture to be used for real-time simulations as well as post-simulation analysis. This also provides the greatest possible resolution for the event histories, which implies that any quantity that could have been calculated during the simulation can also be calculated by a post-simulation analysis of the event history, without any loss of accuracy. It also eliminates the need for the WriteStatus event in the legacy software. The only constraint imposed by this design refinement is that the simulation objects associated with the events must be copied before being included in the simulation history, to protect them from further changes of state as the simulation proceeds. This constraint is easy to meet because the process of writing the contents of an event object to a history file will implicitly make the required copy.

The prototyping effort also exposed a design issue—should null events appear in the event queue? A null event is one that does not affect the state of the simulation, such as a MoveUpdateObj event for an object that is currently stationary. The prototype version adopted the position that such events should not be put in the event queue, since this corresponds to scheduling policies in the legacy system, and appears at first glance to improve efficiency.

Our experience with the development of the prototype suggests that this decision complicates the logic and may not in fact improve efficiency. In particular, the process *create_new_events* could be eliminated from the Janus subsystem (Figure 6) if we allowed null events. This process scans all simulation objects once per simulation cycle to determine if any dormant objects have become active, and if so, schedules events to handle their new activities. The alternative is to have the constructor of each kind of simulation object schedule all of its initial events, and to have each event handler specify the time of next instance of the same event even if there is nothing for it to do currently. Handlers might still set the time of its next event to NEVER in the case of a catastrophic kill: however this is reasonable only if it is impossible to repair or restore the operation of the units that have suffered a catastrophic kill.

The reasons why this design change may improve efficiency in addition to simplifying the code are that:

(1) the check for whether a dormant object has become active is done less often—once per activity of that object, rather than once per simulation cycle.

(2) executing a null event is very fast—a few instructions at most, so the "unnecessary" null events will not have much impact on execution time, and

(3) the computation to find and test all simulation objects periodically would be eliminated.

Our recommendation is to allow null events in the event queue, and to explicitly schedule every kind of event for every object unless it is known that there cannot be any non-empty events of that type in any possible future state of the object. For example, under the proposed scheduling policy, immobile or irrecoverably damaged objects would not need to schedule future MoveUpdateObj events, but those that are currently at their planned positions would need to do so, because a change of plan would cause them to move again in the future, even though they are not currently moving.

6. Conclusion

Our experience in this case study suggests that prototyping can be a valuable aid in the re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed.

In particular, we found that constructing even a very thin skeletal instance of the proposed new architecture raised many issues and enabled us to correct, complete, and optimize the architecture for both simplicity and performance. This was done before the architecture had grown into a maze of dependent designs and implementation details. Consequently, the changes could be realized without incurring the large cost and time delays typically encounted later in the development.

The computer-aided prototyping tools in the CAPS system enabled us to do this with a minimal amount of coding effort. The bulk of the code was generated automatically, enabling us to concentrate on system structuring issues, to consider and evaluate various alternatives, and to improve the design while doing detailed manual implementation for only a few pages of critical code.

The object models produced in this project have proven invaluable to the contractors during code implementation phase of the US Army TRAC HLA Warrior project and will be vital to the National Simulation Center Spectrum project. Additionally, our efforts will also benefit other simulation developers. TRAC-Monterey sent the class design to Combat 21 (CB21) developers at White Sands. CB21 was able to save time and money by reusing the object models and came up with a design that looks remarkably like ours (although much larger). The OneSAF developers will look at the CB21 class design and reuse as much as possible.

Acknowledgments

This research was supported in part by the U.S. Army Research Office under grant number 35037-MA and in part by a grant from the U.S. Army Training and Doctrine Analysis Command.

References

1. Berry, D. 1999. Formal methods: the very idea—some thoughts about why they work when they work. *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems* pp. 9–18.
2. Biggerstaff, T. 1990. Design recovery for maintenance and reuse. *IEEE Computer* 22(7): 36–49.
3. Biggerstaff, T. 1990. Human-oriented conceptual abstractions in the re-engineering of software. *Proceedings of the 12th ICSE* p. 120.
4. Gall, H., and Klösch, R. 1993. Capsule oriented reverse engineering for software reuse. *Proceedings of the 4th European Software Engineering Conference* pp. 418–433.
5. Jacobson, I., and Lindstrorm, F. 1991. Re-engineering of old systems to an object-oriented architectures. *Proceedings of OOPSLA91* pp. 77–83.
6. *Janus Version 6 User's Manual.* 1995. Simulation, Training & Instrumentation Command, Orlando, Florida.
7. *Janus 3.X/UNIX Software Programmer's Manual* 1993. Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov.
8. *Janus Version 6 Database Management Program (CSDATA) Manual.* 1995. Simulation, Training & Instrumentation Command, Orlando, Florida.
9. *Janus 3.X/UNIX Software Design Manual.* 1993. Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov.
10. Luqi, and Ketabchi, M. 1988. A computer-aided prototyping system. *IEEE Software* 5(2): 66–72.
11. Luqi, 1996. System engineering and computer-aided prototyping. *Journal of Systems Integration—Special Issue on Computer Aided Prototyping* 6(1): 15–17.8.
12. Pimper, J., and Dobbs, L. 1988. *Janus Algorithm Document (Version 4.0)* Lawrence Livermore National Laboratory, California.
13. Rieger, L., and Pearman, G. 1999. Re-engineering legacy simulations for HLA-compliance. *Proceedings of the Interservice/Industry Training* Simulation and Education Conference (I, ITSEC), Orlando, Florida, December.
14. Shing, M., Luqi, Berzins, V., Saluto, M., and Williams, J. 1999. Re-engineering the Janus Combat Simulation System. *Tech. Report NPSCS-99-004* Monterey, CA: Computer Science Department, Naval Postgraduate School, January.
15. *TAE Plus C Programmer's Manual (Version 5.1).* 1991. Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April.

Object-oriented modular architecture for ground combat simulation*

V. Berzins, M. Shing, Luqi
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943
{berzins, mantak, luqi}@cs.nps.navy.mil

M. Saluto
EECS Department
United States Military Academy
West Point, NY 10996
dm5447@exmail.usma.army.mil

J. Williams
JSIMS Joint Program Office
12249 Science Dr., Suite 260
Orlando, Fl 32826
julian_williams@jsims.mil

Abstract

This paper addresses the need to modernize the software of the US Army Janus(A) combat simulation system into a maintainable and evolvable structure. It describes the effective use of computer-aided prototyping techniques for re-engineering the legacy software and presents the resultant object models and modular architecture for the existing Janus(A) system. The object models produced in this project have proven invaluable to the contractors during code implementation phase of the US Army TRAC HLA Warrior project and beneficial to other simulation developers.

1. Introduction

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. Legacy systems embody substantial institutional knowledge, which include basic and refined requirements, design decisions, and invaluable advice and suggestions from domain users that have been implemented over the years. To effectively use these assets, it is important to employ a systematic strategy for continued evolution of the current system to meet the ever-changing mission, technology and user needs. However, knowledge embedded in these systems is difficult to recover after many years of operation, evolution, and personnel change. These software systems were originally written twenty or more years ago using what many now view as an archaic and ad-hoc methodology. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that now must be changed over large areas of the code. Re-

* This research was supported in part by the U.S. Army Research Office under contract # 35037-MA and in part by the U. S. Army Training and Doctrine Analysis Command.

engineering has frequently been proven to be more cost effective than new development and is also known to better promote continuous software evolution.

Software re-engineering can be defined as the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer. Such improvements often take the form of increased or enhanced functionality, better maintainability, configurability, reusability, and/or other software engineering goals. This process involves recovering existing software artifacts from the system and then re-organizing them as a basis for future evolution of the system. The re-engineering of procedural legacy software into modern object-oriented architectures introduces certain complexities into the software analysis process. Since typical legacy systems were not originally designed and implemented using an object-oriented approach, the products of reverse engineering, such as requirements or design specifications, will probably reflect a functionally based approach. As a result, some form of "transformation" of resultant information is necessary in order to use the specifications. Once a realizable specification based on the transformed object-oriented models is obtained, it is often very difficult to quickly determine if the specification is a true representation of the desired requirements. Since legacy systems are usually re-engineered only when the existing systems need some kind of improvement, it is unlikely that the initial version of the reconstructed requirements adequately reflects current user needs. Prototyping provides a means to validate new system requirements while simultaneously enabling prospective users to get a brief feel for aspects of the proposed system. It is a well-established approach that can be highly effective in increasing software quality [13]. When used in conjunction with conducting a major re-engineering effort, prototyping can be extremely useful in assisting in many areas of software modification, validation, risk reduction, and the refinement of user requirements.

This paper addresses the need to modernize the software of the Janus(A) systems into a maintainable and evolvable structure. It describes the effective use of computer-aided prototyping techniques for re-engineering the legacy software [14] to develop an object-oriented modular architecture for the Janus combat simulation system [16]. Janus(A) is a software-based war game that simulates ground battles between up to six adversaries [7]. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of a large number of FORTRAN modules (1918 FORTRAN routines, 115 C routines, and a total of 393,000 lines of source code). The FORTRAN modules are organized as a flat structure and interconnected with one another via 129 FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create a base-line object-oriented architecture that supports existing and required enhancements to Janus functionality.

2. Reverse Engineering

The re-architecturing process adapted in the project consists of 3 major phases: reverse engineering, object-oriented design and design validation via prototyping (Figure 1).

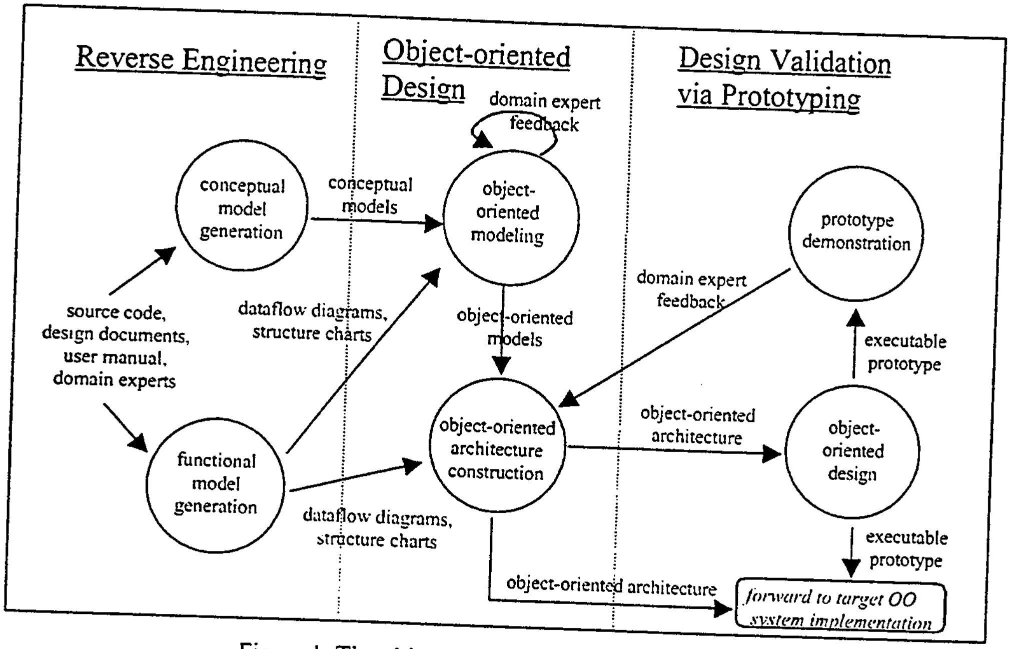

Reverse Engineering
Object-oriented Design
Design Validation via Prototyping
domain expert feedback
conceptual model generation
conceptual models
object-oriented modeling
prototype demonstration
domain expert feedback
source code, design documents, user manual, domain experts
dataflow diagrams, structure charts
object-oriented models
executable prototype
object-oriented architecture construction
object-oriented architecture
object-oriented design
functional model generation
dataflow diagrams, structure charts
executable prototype
object-oriented architecture
forward to target OO system implementation

Figure 1. The object-oriented re-architecturing process

The first phase is reverse engineering. Input to this phase includes the legacy source code, design documents, user manuals, and information from domain experts. Since the goal of the re-engineering effort is to duplicate the functionality of the existing system within a modular, extensible architecture and to reuse the domain concepts, models and algorithms rather than the existing code, we should avoid including any requirements/constraints that are consequences of the FORTRAN implementation. The best places to extract domain concepts from the existing system are the user manuals and the database management system manuals. These manuals were written using the lingo of the user community and should be relatively free of implementation details. We found the JANUS Data Base Management Program Manual [8] particularly useful because it contains detailed information on what kind of data are needed to model the battle field and how they are organized (logically) in the database. The top-level structure of the database is shown in Figure 2.

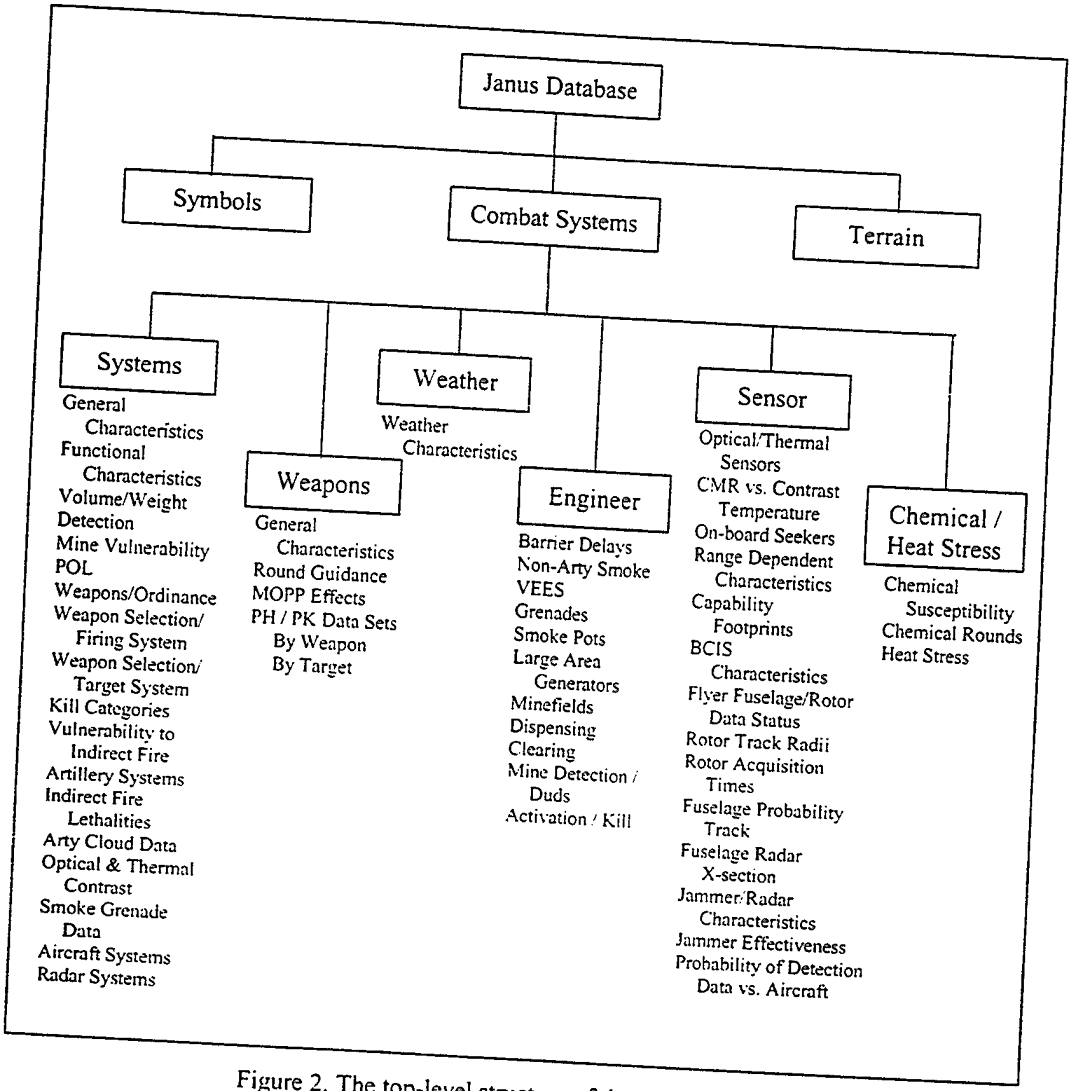


Figure 2. The top-level structure of the Janus Database

Not shown in Figure 2 are the interdependencies between the data, where data entered in one category affect directly or indirectly the data in other categories. For example, the barrier delays of the Engineer Data depend on specific weather condition specified by the Weather Data and system functional characteristics of the System Data of the database. The overall network of interdependencies is highly complex and can only be understood through construction and analysis of the functional model of the existing Janus software.

Analysis of 393K lines of legacy code is a daunting but inescapable part of the process. We recoiled from the magnitude of this effort in the beginning of the project and relied on information contained in the Janus manuals. In hindsight, it was a mistake that slipped the schedule of the project by several months. While these documents helped us get started because they contained higher level information and were much shorter than the code, they were much older and contained outdated information. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross-fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively.

Using manual techniques augmented with simple UNIX shell commands, we were able to walk through the code and get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [6] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs and develop functional models from the data flows. We used the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School, to assist in developing the abstract models [12]. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

We also had a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. These meetings were indispensable because they gave us information that was not present in the code. Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [1]. Domain analysis has been identified as an effective technique for software re-engineering [15]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

3. Object-Oriented Design

Next, we developed object models and architecture of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [4]. This was probably the most difficult and most important phase. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this phase, we used our knowledge of object-oriented analysis and applied the OMT techniques [17] and the UML notations to create the classes and associated attributes and operations [18]. This was a crucial phase because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software.

Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [5]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

The re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core data elements and the object-oriented architecture for the Janus System. We presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center project. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced in [9] that the involvement of domain experts is critical for nontrivial re-engineering tasks.

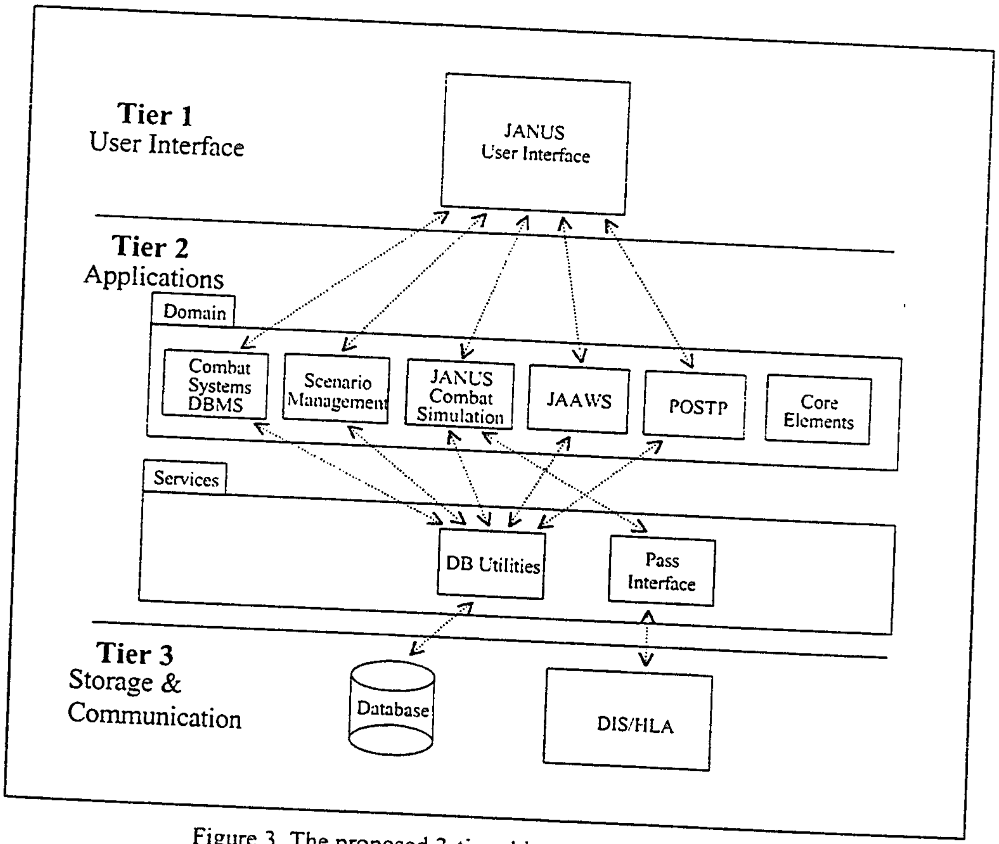


Figure 3. The proposed 3-tier object-oriented architecture

Early involvement of the stakeholders in the simulation community also pays off in the long run. Both the National Simulation Center and Combat21 projects were able to save time and money by reusing our work and came up with designs that look remarkably like ours (although much larger). Now, OneSAF developers have been directed to look at the Combat21 class design and reuse as much as possible. So, our efforts have directly benefited other simulation developers.

Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 3). We extracted most of the data and operations from the existing Combat System DBMS, Scenario Management, Janus Combat Simulation, JAAWS and POSTP subsystems and encapsulated them as simulation objects in the Core Elements package, leaving only application specific control codes that use the simulation objects in each of these five subsystems. Figures 4 and 5 show the top level class structures of the object models of the core elements. Details of the associated attributes and operations can be found in [2, 20] and are omitted from these diagrams due to space limitations.

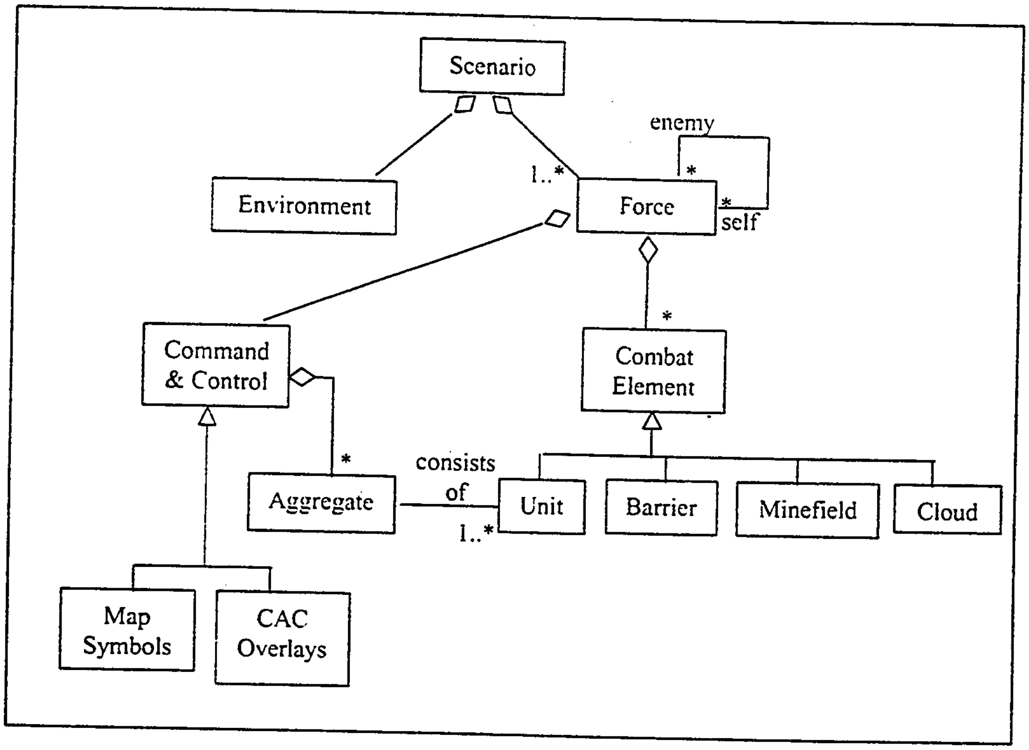


Figure 4. The top-level structure of the Janus Core Elements Object Model

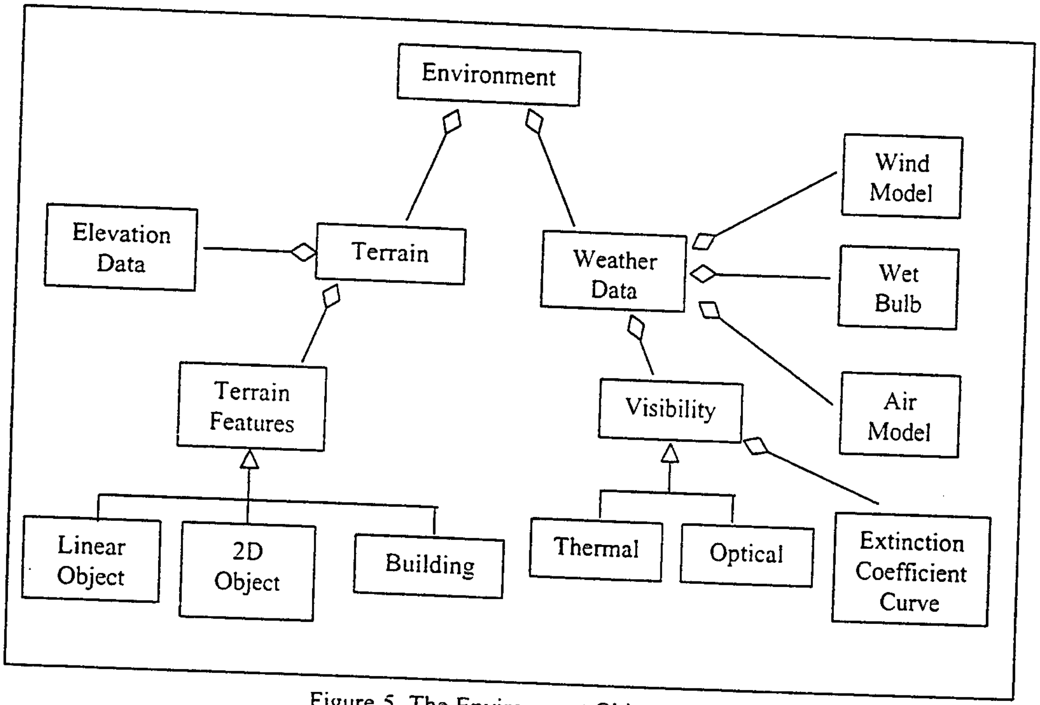


Figure 5. The Environment Object Class

Central to the Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advance the game clock to the scheduled time of the event and perform that event. The existing Janus Simulation System uses 17 different categories to characterize the events. RUNJAN then handles these 17 events using the following event handlers:

1) DoPlan - Interactive Command and Control activities
2) Movement - Update unit positions
3) DoCloud - Create and update smoke and dust clouds
4) StateWt - Periodic activity to write unit status to disk
5) Reload - Plan and execute the direct fire events
6) Intact - Update the graphics displays
7) CntrBat - Detect artillery fire
8) Search - Update target acquisitions, choose weapons against potential targets, and schedule potential direct fire events
9) DoChem - Create chemical clouds and transition units to different chemical states
10) Firing - Evaluate direct fire round impacting and execute an indirect fire mission
11) Impact - Evaluate and update the results of an indirect round impacting
12) Radar - Update an air defense radar state and schedule a direct fire event for "normal" radar
13) Copter - Update a helicopter states

14) DoArty - Schedule an indirect fire mission
15) DoHeat - Update units' heat status
16) DoCkpt - Activity to perform automatic checkpoints
17) EndJan - Housekeeping activity to end the simulation

The legacy event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation Subsystem was to distribute the event handling functions to individual objects. However, many of the current event handler categories contained redundant code and did not seem to be very coherent with respect to the class hierarchy we created. For example, the set of event handlers used to simulate the activities of a particular unit to search for targets, select weapons, prepare for a direct fire engagement, and then execute that direct fire engagement differs depending upon whether the unit has a normal radar, special radar, or no radar at all. The legacy Janus Simulation System uses the Radar event handler to carry out the entire procedure if the unit has normal radar. However, it uses the Search, Radar, and Reload event handlers to carry out the procedure if the unit has special radar. Finally the system uses the Search and Reload event handlers to conduct the procedure if the unit has no radar at all. We conjecture that this lack of uniformity is due to a series of software modifications made by different people at different times without full knowledge of the software structure.

It was necessary to redefine some event categories in order to reduce interdependencies between the event handlers, to factor simulation behavior into more coherent modules, to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Moreover, the Janus system was originally designed to work in isolation, and has since been adapted to interact with other simulation systems. Interactions between the simulation engine and the world modeler (the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation.

The new architecture uses an explicit priority queue of event objects to schedule the simulation events. We were able to reduce the total number of event handlers needed in the simulation, from 17 to 14, by eliminating identified redundant code (Figure 6). The 14 remaining event handlers are as follows:

1) DoPlan - Interactive Command and Control activities
2) MoveUpdateObj – Moves and update the objects in the simulation
3) Search – Searches for potential targets based on the detection devices available to the objects
4) ChooseDirectFireTargets – Once search is complete chooses best target to engage. In future simulations, implementations may allow users to choose targets
5) CounterBattery – Simulates counter battery radar to find potential targets
6) DoDirectFire – Executes direct fire events and updates ammunition status
7) DoIndirectFire – Executes indirect fire events and updates ammunition status
8) ImpactEffects – Calculates results of round impacting

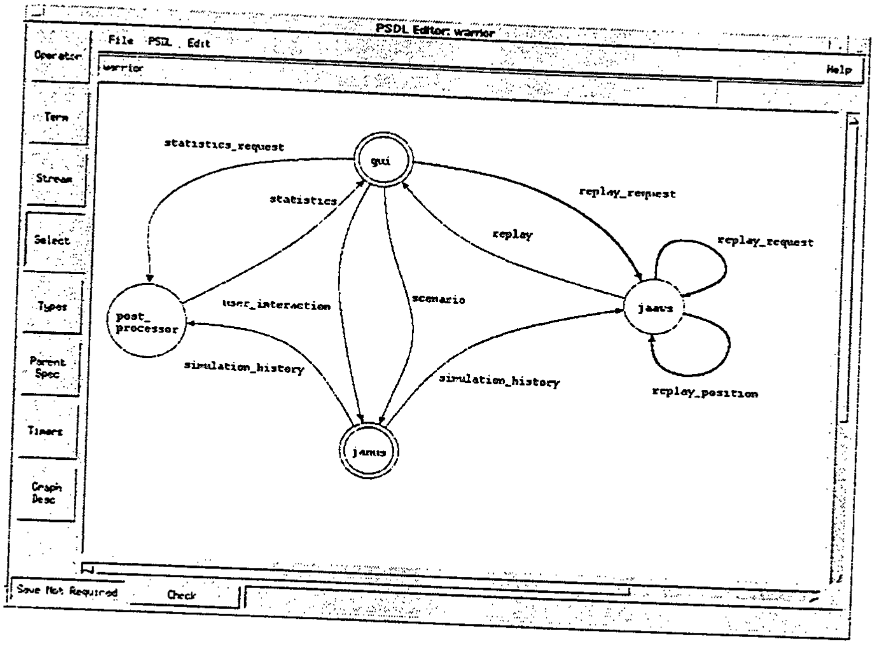
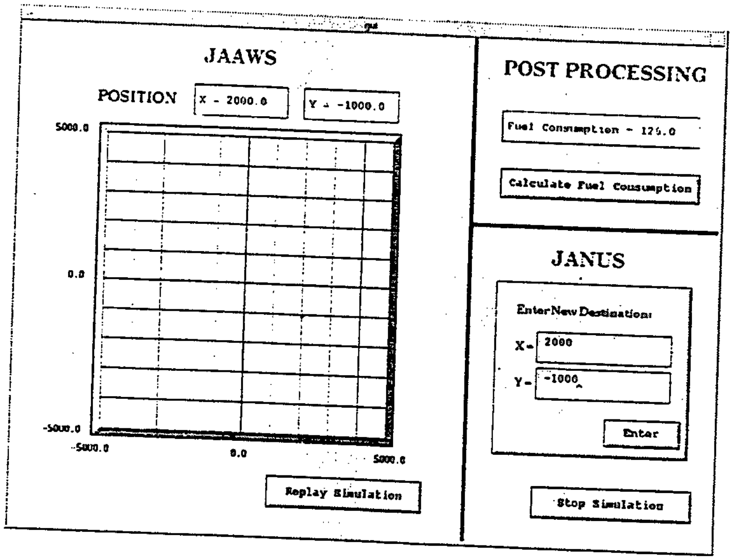


Figure 8. Top-level decomposition of the executable prototype



POSITION X = 2000.0 Y = -1000.0

Fuel Consumption = 125.0

EnterNew Destination:

X= 2000

Y= -1000

Figure 9. The Graphical User Interface of the executable prototype

5. Conclusions

Our conclusion is that substantial and useful computer aid for re-engineering is possible at the current state of the art. Human analysts must also play an important part of the process because much of the information needed to do a good job is not present in the software artifacts to be re-engineered. Success depends on cooperation between skilled people and appropriate software tools.

The missing information needed for re-engineering is related to deficiencies of the current system at all levels, from requirements through design and implementation. Thorough and accurate knowledge of these deficiencies is crucial for success. The clients never want the re-engineered system to have the exactly same behavior as the legacy system - if they were satisfied, there would be little motivation to spend time, effort, and resources on a re-engineering project. Even if a system is being re-engineered for the ostensible goal of porting to different hardware, the desired behavior at the interface to the hardware and systems software will be different.

In practical situations, the requirements for the re-engineered system are different from those for the legacy system. Key parts of the requirements for the new system are often missing or incorrect on the legacy documents. Some of that information is present only in the minds of the clients, often fragmented and scattered across members of many different organizations. Communication is a large part of the process, and that communication cannot be automated away, although it can be enhanced by appropriate use of prototyping.

Uncertainties about the true requirements play a central role in both re-engineering and the development of new systems. We therefore hypothesized that prototyping could play a valuable role in re-engineering efforts. Our experience supports that hypothesis.

We also found that prototyping can contribute substantially to the process of inventing, correcting, and refining the conceptual structures on which the architecture of the new system will be based. Most legacy systems are too complicated for individuals to understand. We found that constructing even a very thin skeletal instance of the proposed new architecture raised many issues and enabled us to correct, complete, and optimize the architecture for both simplicity and performance. (See [3] for lessons learned from the prototyping effort.) This was done before the architecture had grown into a maze of dependent designs and implementation details. Consequently, the changes could be realized without incurring the large cost and time delays typically encounted later in the development.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. The UML interaction diagrams lack the preciseness to support automatic code generation for the executable prototype. Such weakness can be remedied by the use of the prototype language PSDL [10, 11] and the CAPS prototyping environment, which provide effective means to model the system's dynamic behavior in a form that can be easily validated by user via prototype demonstration.

8. References

[1] D. Berry, Formal Methods: The Very Idea, "Some Thoughts About Why They Work When They Work," Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems, 1998, pp. 9-18.

[2] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, *Re-engineering the Janus(A) Combat Simulation System*, Technical Report NPS-CS-99-004, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.

[3] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, "Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping," to appeare in the journal *Design Automation for Embedded Systems*. A preliminary version of the paper also appeared in *Proceedings of the 10th IEEE International Workshop in Rapid Systems Prototyping*, Clearwater Beach, Florida, 16-18 June 1999, pp. 216-221.

[4] O. Bray and M. Hess, "Reengineering a Configuration-Management System," *IEEE Software*, Vol. 12, No. 1, Jan. 1995, pp. 55-63.

[5] V. Cabaniss, B. Nguyen and J. Moregenthaler, "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems," *IEEE TSE*, Vol. 24, No. 7, July 1998, pp. 534-558.

[6] *Janus 3.X/UNIX Software Programmer's Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[7] *Janus Version 6 User's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[8] *Janus Version 6 Data Base Management Program Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[9] S. Jarzabek and P.K. Tan, "Design of a Generic Reverse Engineering Assistant Tool," *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, 1995, pp. 61-70.

[10] B. Kraemer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, May 1993, pp. 453-477.

[11] Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 14, No.10, October 1988, pp. 1409-1423.

[12] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, Vol. 5, No. 2, 1988, pp. 66-72.

[13] Luqi, "System Engineering and Computer-Aided Prototyping," *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, Vol. 6, No. 1, 1996, pp.15-17.

[14] Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo and B. Shultes, "The Story of Re-engineering of 350,000 Lines of FORTRAN Code," *Proceedings of the 1998 ARO/ONR/NSF/ DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, 23-26 October 1998, pp. 151-160.

[15] M. Moore and S. Rugaber, "Domain Analysis for Transformational Reuse," *Proceedings of 4th Workshop on Reverse Engineering*, IEEE Computer Society, 1997, pp. 156-163.

[16] L. Rieger and G. Pearman, "Re-engineering Legacy Simulations for HLA-Compliance," *Proceedings of the Interservice/Industry Training. Simulation and Education Conference (I/ITSEC)*, Orlando, Florida, December 1999.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenzer, *The Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[18] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley. Reading, MA, 1999.

[19] *TAE Plus C Programmer's Manual (Version 5.1)*. Prepared for: NASA Goddard Space Flight Center, Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

[20] J. Williams and M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X*, master's thesis, Naval Postgraduate School, Dept. of Computer Science, Monterey, CA, March 1999.

Static Analysis for Program Generation Templates[1]

Valdis Berzins
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

This paper presents an approach to achieving reliable cost-effective software via automatic program generation patterns. The main idea is to certify the patterns once, to establish a reliability property for all of the programs that could possibly be generated from the patterns. We focus here on properties that can be checked via computable static analysis. Examples of methods to assure syntactic correctness and exception closure of the generated code are presented. Exception closure means that a software module cannot raise any exceptions other than those declared in its interface.

1. Introduction

Our goal is to provide cost effective means for creating reliable software. We are addressing the issue by improving the technology for automatic software generation, with particular attention to reliability issues.

We take a domain specific view of this process: a domain is a family of related problems addressing a common set of issues. A domain analysis identifies the problem and issues, formulates a model of these, and determines a corresponding set of solution methods. Users of the proposed computer-aided software generation system describe their particular problem using a domain specific problem modeling language that provides concrete representations of problems in the domain. The system then automatically determines which solution methods are applicable, customizes them to the specific problem instance described using the modeling language, and then automatically generates a program that will solve the specified problem.

We seek to provide tool support for the above process that can be applied to many different problem domains, and that can generate code in any programming language. Therefore we seek uniform and effective methods for generating software generators of the type described above, given definitions of the problem modeling language, the target programming language, and the roles for synthesizing solution programs. A simple architecture for this process is shown in Figure 1.

The specific goals of this paper are: (1) to provide a simple example of a language for expressing software patterns that are specific enough to be used as synthesis rules and (2) to provide examples of static rules in this language. We address the problems of certifying that all programs which can be generated from a given set of rules: (1) are syntactically correct and (2) will not raise any exceptions other than those explicitly specified in an interface description.

This is a step towards a coordinated system of static and dynamic checks, to be performed on program synthesis rules. Our hypothesis is that the most cost effective way to improve software quality is to systematically improve and certify the rules used to generate a domain-specific software generator. This approach directly addresses the issue of correctly implementing given software requirements. It also indirectly addresses the issue of getting the right requirements, because it should eventually enable rapid prototyping of product quality systems by problem domain experts, who need not be software experts. If the requirements are found to be inappropriate, the domain experts will simply update the problem models and regenerate a new version of the solution software.

We will refer to the software generation patterns as templates. Our rationale for the claim of cost effectiveness is that the benefits of quality improvements to the templates can be extended to all past and future applications of the generators - by regenerating the generator using the improved templates and then regenerating the past applications. The regeneration process can be completely automated, thereby reducing labor costs, eliminating a source of random human errors, and speeding up the process of repairing a known fault throughout a large family of software systems.

[1] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA, and in part by DARPA under contract #99-F759.

The relation to the theme of this workshop is that fast moving scenarios can be addressed by automatically generating new variants of the software that reflect changing issues in the problem domain. Our approach should reduce the explicit quality assurance efforts needed each time the software is changed. By amortizing the quality assurance effort applied to the template over many applications of the same templates, we can reduce quality assurance costs. The benefits increase with the number of systems generated from the same templates.

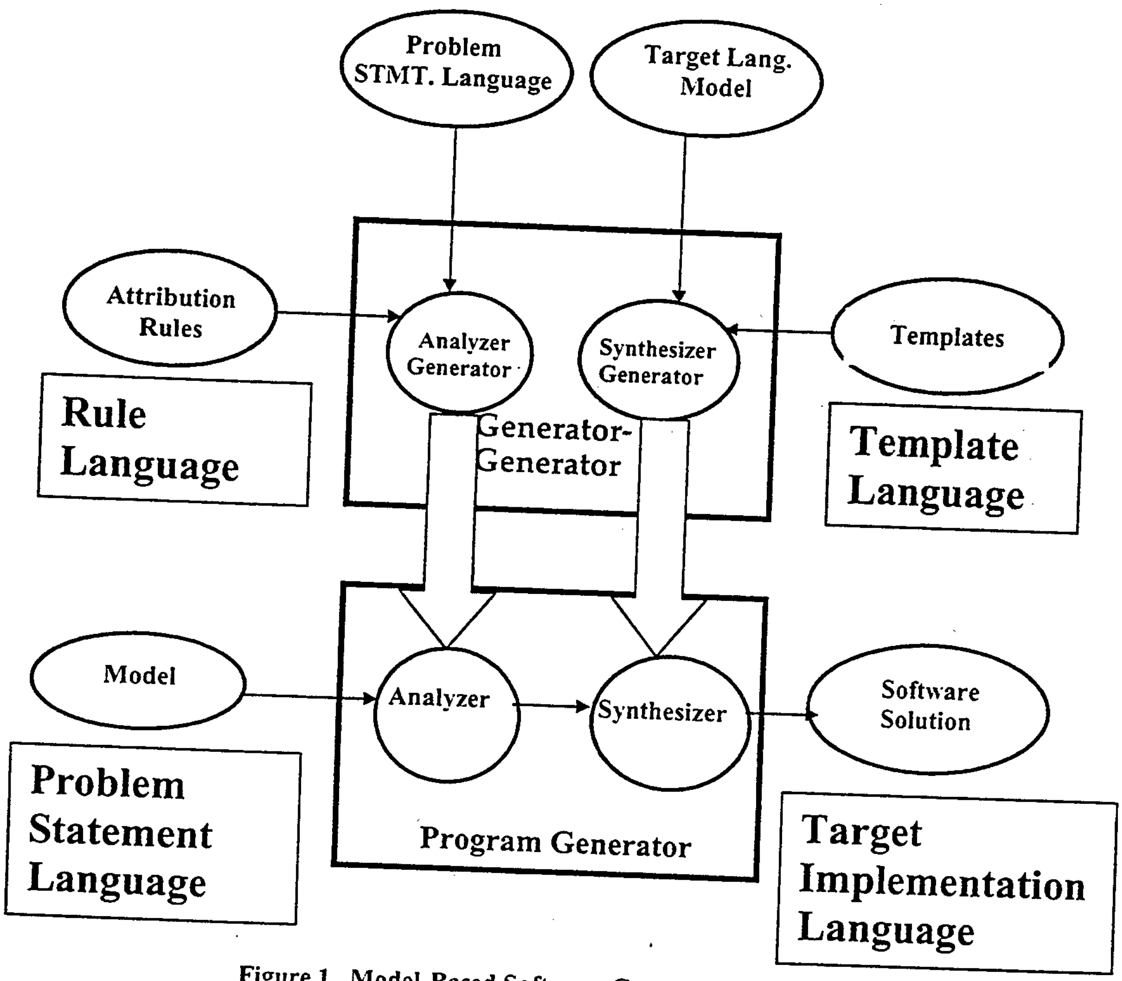


Figure 1. Model-Based Software Generator Architecture

This paper focuses on static checks that can be completely automated. Our research is also addressing testing and debugging of program synthesis rules and proofs of rule properties that require human assistance with deeper reasoning. These efforts are outside the scope of the current paper, which is organized as follows:

- Section 2 formalizes software generation patterns and defines a uniform construction to obtain a template language for any target programming language.
- Section 3 describes methods for statically certifying syntactic correctness generated code, and gives an example.
- Section 4 does the same for analysis of exceptions.
- Section 5 contains comparisons to previous work
- Section 6 presents conclusions.

2. Template Languages

The purpose of a template language is to define software synthesis patterns for a given target language. We create such languages based on a functional object model of code generation templates. We take a functional (i.e. side-effect-free) approach because this simplifies the algebraic basis of the approach and supports effective static analysis methods such as those presented in Section 3 and 4.

We view template languages as extensions of the corresponding target programming languages. Because many different programming languages are created, we will need many different template languages. However, all of these can be defined at once by providing uniform construction such as that shown in Figure 2.

This is a very simple construction, but it is very expressive. In addition to providing substitution of actual values for generic parameters, as in the generic units of Ada and the templates of C++, our construction includes conditionals that are evaluated at code generation time, and the ability to invoke other templates. Recursion is included.

```
Template_language = {template, formal_def, template_expression}

DEF_TEMPLATE(id[template], type, seq[formal_def]. template_expression):
        template     -- where type ° ∈ target_language

DEF_FORMAL(template_parameter, type): formal_def
        -- declares the type of a formal parameter

template_parameter < {id[any], template_expression}
IF(template_expression, template_expression, template_expression):
        template_expression
APPLY(id[template], seq[template_expression]): template_expression

template_expression < target_language
```

Figure 2. Template Abstract Syntax

The construction depends heavily on the use of inheritance in object-oriented modeling of programming languages. The situation is illustrated in Figure 3.

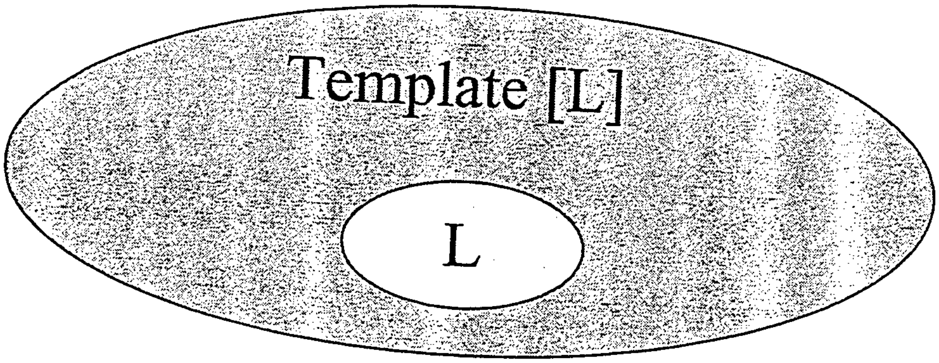


Figure 3. Generic Template Language

In object-oriented modeling, class-wide types[2] are viewed as open and extensible. Specifically, each time we add a subclass with a new constructor, we add more instances to the class-wide type, thus extending its value set.

We model the abstract syntax of a language using a type for each kind of semantic entity. In a properly constructed abstract syntax, there should be one such type for each non-terminal symbol. Each constructor of these types corresponds to a production of the grammar. Subclass relationships, denoted by "≤", specify that every instance of the subclass is also an instance of the parent class. Multiple inheritance is allowed. For example, in line 6 of Figure 2 says that every template parameter is a kind of identifier, and also is a kind of template expression. This kind of subclass relationship is used to incorporate reusable types in a library of programming language building blocks, such as identifiers, and to specialize reusable concepts to the application, such as template expression. If T is a type and S is a set of types, T<S means T is a subclass of each element of S. This represents multiple inheritance.

[2] This is Ada 95 terminology. The instances of a class wide type include its direct instances and those of all its subclasses, transitively.

Subclassing is also used to interface between a target programming language and its extensions. In Figure 2, "target-language" denotes the set of types comprising the abstract syntax of the target language. Figure 4 shows a very simple example of a target language that illustrates how this works.

```
target_language = (stmt, exp)

assign(var, exp): stmt
if(exp, stmt, stmt): stmt

integer < exp    -- integer literals
var < {id[any], exp} -- program variables
apply(id[function], seq[exp]): exp -- operations

subtype rule:  x < y ==> id[x] < id[y] where x, y ∈  type
```

Figure 4. Example: Micro Target Language

The example in Figure 5 defines a code generation pattern that embodies Newton's method for polynomial evaluation, which is optimal in terms of number of evaluation steps needed. This is a very simple example of a code generation pattern that is nevertheless realistic, because it embodies a solution method. The example also illustrates the use of all the constructs in the template language. We use infix syntax for the exp constructors * and + to improve legibility (e.g. x*y is short for the term apply(*, x, y)).

An additional benefit of considering the abstract syntax to be an algebra rather than a tree is that we can used well-studied transformation rules. In particular we can associate equational axioms with the programming language types that define normal forms. Figure 5 illustrates the use of such axioms as rewrite rules that simplify the code produced by the generator in a follow-on normalization process. This is one way to incorporate optimizations into the program generation process, which is useful for unconditional transformations.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
    -- c contains coefficients of a polynomial, lowest degree first
    IF not (is_empty (c) )  -- use operations of boolean and seq
    THEN v * (evaluate_polynomial (v, rest(c))) + first (c)
    ELSE 0
END TEMPLATE

Template application evaluate-polynomial(x, [1, 2, 3]) generates
    x * (x * (x * 0 + 3) ÷ 2) + 1

Normalization with integer rules i * 0 = 0, i + 0 = i reduces to
    x * (x * 3 + 2) + 1
```

Figure 5. Example: Generation Pattern

Code generation using the template language is a very much like evaluation in a functional programming language with call-by-value semantics. Analysis of templates can take advantage of equational reasoning, substitution, and structural induction. The limitation to primitive recursion facilitates the latter. The recursion in the example is structural because **rest** is a partial inverse for the sequence constructor **add** (i.e. rest(add(x, s)) = s).

3. Syntactic Correctness of Generated Code

We treat the abstract syntax structures of the target language as the values of the abstract data types representing the programming language. We require these types to provide a pretty printing operation that outputs such objects as text strings according to the concrete syntax of the target language, with a readable format. Establishing correctness of these pretty printing operations is straightforward, and in fact their implementations can be generated from an appropriately annotated grammar for the concrete syntax.

Given trusted pretty printing operations for the object model of the target language, syntactic correctness of the output reduces to the type-correctness of the ground terms generated by the evaluation of the templates. This can be checked using a simple type system for the template language and conventional type checking methods. Note that we are referring to the types associated with the signatures of the constructors in the object model of the target programming language, rather than the types within the target programming language, which may not even be a typed language. The process is illustrated Figure 6. The computed type annotations are shown in *italics*. The type annotations associated with the implicit induction step, where the type signature of the template itself is used, is highlighted in ***bold italics***. The indentations of the type annotations reflect the structure of the derivation.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
  IF not (is_empty (c : seq[integer]  ) : boolean ) : boolean
  THEN  + (  * ( v
                           evaluate_polynomial                              : var,
                               (v                              : var,
                                 rest(c: seq[integer] ) : seq[integer]) : exp
                          first (c: seq[integer] )                           ) : exp
                                                                               : integer
                   --term form of v* evaluate_polynomial (v, rest(c)) + first (c)   ) : exp
  ELSE 0
END TEMPLATE                                                                   : integer
```

Types conform because `integer < exp and var < exp`

Relevant signatures: +(exp, exp) :exp, *(exp, exp) :exp,
first(seq[T]): T, rest(seq[T]): seq[T],
is_empty(seq[T]): boolean, not(boolean): boolean

Figure 6. Example: Syntactic Correctness of Generated Code

Note that induction has been carried out implicitly, as a routine step of the type checking calculation. This is sufficient to establish partial type correctness of the templates, which implies syntactic correctness of all code that could be generated by the template. it does not automatically guarantee total correctness, because we still have the possibility that evaluation of the template might fail to terminate.

Total correctness is established by the type check if we check that all recursions are primitive. The example satisfies this condition because rest is a partial inverse of the compound sequence constructor; rest(add(x,s)) = s. This means that the induction is in fact structural, and hence that evaluate_polynomial is total. Thus the template will produce syntactically correct code for all input values that conform to the type signature of evaluate_polynomial.

We note that given declarations of the target language constructors that define the abstract syntax and the corresponding partial inverse operations, it is straightforward to automatically check that all recursive calls are primitive with respect to any given parameter position. This implies that structural induction can be applied uniformly and completely automatically in this context. Furthermore, our experience suggests that structural recursions are sufficient to define the code generation templates needed in practice, and that template designers can live within the restriction to structural recursions without undue hardships.

4. Exception Closures for Generated Code

One common source of software failure is unhandled exceptions. This section explains a method for certifying that all programs generated from a given template cannot generate any unhandled exceptions when placed in a context that handles a specified set of exceptions.

Our approach is to refine the type system to record the set of exceptions that might be raised by the evaluation of any expression of the target language. A similar structure can be used to analyze the set of exceptions that might be raised by execution of a statement of the target language.

The refinement replaces the single target language type exp with a parameterized family of types exp[set[exception]]. The intended interpretation of this type structure is that evaluation of an expression of type exp[S] might raise an exception e only if $e \in S$. Since we do not require all exceptions in S to be producible, this family of types has a rich subclass structure defined by the following relation:

$$S1 \subseteq S2 \Rightarrow \text{exp}[S1] \leq \text{exp}[S2]$$

The type signatures of an operation are specified explicitly for argument expression type that cannot raise any exceptions, and are extended to all other types by the following rule, which describes the essential pattern for propagating exceptions:

$$F(\text{exp}[\varnothing]) : \text{exp}[S1] \Rightarrow f(\text{exp}[S2]) \text{: exp}[S1 \cup S2]$$

The rule for operations with multiple arguments is similar. Similar rules apply to language constructs representing exception handlers. Exception handlers follow rules of the form

$$\text{(TRY exp}[S1] \text{ CATCH e USE exp}[S2]\text{): exp}[(S1\text{-}\{e\}) \cup S2].$$

Figure 7 shows the exception analysis for our running example. The parts added to the version in Figure 6 are <u>underlined</u>.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp [{ovfl}]
 IF not (is_empty (c: seq[integer]boolean boolean
   THEN +(*(v: var
            evaluate_polynomial(v: var,
               rest(c: seq[integer]seq[integer ): exp [{ovfl}]
            first (c: seq[integer]integer exp[{ovfl}]
    -- term form of v * evaluate_polynomial (v, rest(c)) + first (c)
   ELSE 0: integer
END TEMPLATE
```

Types conform because integer < exp [∅] ≤ exp [{ovfl}] and
var < exp [∅] ≤ exp [{ovfl}]

Relevant signatures: +(exp, exp): exp [{ovfl}], *(exp, exp): exp [{ovfl}],
first(seq[T]): T, rest(seq[T]): seq[T], is_empty(seq[T]): boolean, , not(boolean): boolean

Figure 7. Exception Closure of Generated Code

Note that we require the author of the template to specify in the type declaration of a template the set of exceptions the generated expression is allowed to raise. This acts as an induction hypothesis in our exception analysis, which is used when analyzing the recursive call of evaluate-polynomial. It also provides useful information for the user of the generated code.

The analysis shown in the figure establishes a partial exception closure: it guarantees that all expressions generated by the template can at most raise only the exception **ovfl** representing integer overflow.

To establish a total exception closure, we have to address clean termination of the template expansion at program generation time. The primitive recursion check explained in the previous section guarantees there will be no infinite recursions, so that termination is guaranteed. However, for clean termination, we must also check that evaluation of the template will not raise any exceptions at program generation time.

Note that the analysis in Figure 7 addresses run-time exceptions. When viewed as constructors of the abstract syntax, + and * are total operations. Overflow exceptions can occur only when those expressions are evaluated, not when they are constructed.

The sequence operators **first** and **rest** are different: they are partial query methods of the abstract syntax, not total constructors. If applied to an empty sequence, they raise a sequence underflow exception. However, this can occur only at program generation time, not at run time.

To certify clean termination of template at program generation time requires a type refinement to record sets of possible exceptions and an additional kind of type refinement to record domains of partial methods such as **first** and **rest**. We can introduce a subtype nseq[T, S] < seq[T, S] consisting of the nonempty sequences, and refine the signatures of the partial sequence operations first and rest as follows.

first(nseq[T, ∅]): T[∅] , rest(nseq[T, ∅]): seq[T, ∅]
first(seq[T, ∅]): T[seq_underflow], rest(seq[T, ∅]): seq[T, {seq_underflow}]

Type analysis requires a bit of inference in this case, because we have to use the guard of the template language conditional IF together with the rule

s : seq[T, S] and not is-empty (s) ⇒ s: nseq[T, S]

This inference is easy because the guard matches the subtype restriction predicate for nseq[T].

This match did not occur by accident - the purpose of the guard is precisely to ensure that the operations first and rest are used only within their domain of definition. In the interests of being able to produce certifiably robust code, we claim that it would not be unduly burdensome to require that template designers associate domain predicates with all partial operations, and use those domain predicates explicitly in guards whenever they are needed to ensure the partial operators are used within their proper domains of definition. For example, **first** could be associated with a domain predicate

first-ok (seq[T]) : boolean where
first-ok (s) = not (is-empty (s)).

This would enable a fast and shallow analysis of guard conditions to certify absence of exceptions in cases like this. Some such restriction is necessary for practical engineering support because the problem of checking whether an unconstrained guard condition implies the domain predicates of arbitrary guarded partial operations is undecidable.

An alternative is an exception analysis that includes exceptions in the closure even in cases where the guard condition ensures they will never arise. We suggest that it is more practical to handle a common subset of efficiently recognizable forms, and to ask designers to work within the constraints of those recognizable forms. We believe this would be less burdensome than the alternative of manually analyzing the cases where a type check insensitive to guard conditions would nominate exceptions that cannot in fact occur, and that it would lead to a more robust software by making it practical to do complete analysis of exception closures. For example, we could require the example of Figure 7 to be written in a stylized form that looks like the following:

```
IF first-ok (c) and rest-ok (c)
THEN ... first (c) ... rest (c) ...
```

A similar type check would have to be applied to the implementations of **first** and **rest** to ensure that they would in fact terminate cleanly whenever the domain predicates are true.

5. Comparisons to Previous Work

One of our contributions has been to formalize and abstract the idea of a program generation pattern, to make it independent of the details of the target programming language and the process of instantiating the patterns. The purpose of this was to create context in which systematic analysis of program generation patterns becomes possible and in some cases becomes decidable.

Program generation patterns have been evolving for a long time. Macros are an early form of the idea. However, macros are notoriously difficult to analyze, partially because they traditionally operate on uninterpreted text. This makes the connection between macro definitions and the behavior they ultimately denote complicated and potentially very indirect. The macros in LISP are an improvement because they are based on abstract syntax trees rather than characters. However, in this context a second source of complexity becomes apparent: a macro can expand to produce another macro, and the number of expansion steps before the generated source code actually appears is potentially unbounded. This makes the system very difficult to analyze. At the other extreme are the generic units of Ada. These are strongly typed, clearly connected to the abstract syntax of the language, and the results of instantiating them are easy to analyze. However, they do not allow conditional decisions at instantiation time, and are restricted in the sense that the abstract syntax trees of all possible instantiations have exactly the same shape, up to substitution for the formal parameters of the pattern. A language-independent version of the idea can be found in [5], although this appears to be largely text-based.

Another aspect of our approach is to model languages as algebras rather than as abstract syntax trees. A hint of this idea appears in [4], although it is not exploited there for enabling analysis to any significant degree. The work of the CIP group [1] develops this idea further and takes advantage of the reasoning structures that come with the algebraic modeling approach, such as term rewriting and generation induction principles. This suggests extension to a full object-oriented view, which includes inheritance. The Refine system is the earliest context we know of where grammars are treated as object models with potential inheritance structures, although the documentation does not give any hint about the significance of this capability. In this paper we demonstrate the usefulness of algebraic models of syntax with inheritance, for defining language extension transformations that can be applied to all possible target languages.

Another theme is lightweight inference [2]. We have demonstrated that some useful types of static analysis for program generation patterns can be performed via computable and indeed reasonably efficient methods. The processes described here can be implemented using technologies typically used in compilers, such as object attribution rules, they terminate for all possible inputs, and do so in polynomial time. We believe this approach will scale up to large applications, and are currently working out the details to support a tight analysis of the efficiency of the process.

This paper has explored static analysis of meta-programs to check syntactic correctness and exception closure of the generated code. Another kind of static analysis in this family, type checking of meta-programs to ensure the type correctness of the generated code, is considered by another paper in this proceedings [3].

6. Conclusions

We believe that formal models of program generation templates can support a variety of quality improvement processes that can help achieve cost-effective software reliability. This paper has presented a simple example of such a formal model and two such quality improvement processes, certification of syntactic correctness and freedom from unexpected exceptions for all programs that can be generated from a given program generation pattern. We expect the greatest advantages of this approach to be realized when it is applied to realize flexible and reliable systems in a product line approach. This approach should be augmented with systematic methods for domain analysis that culminates in the development of a domain-specific library of solutions embodied in a domain-specific software architecture that is populated with components produced by model-based software generators. When the technology matures, it should become possible for problem domain experts to specify their problem instances in terms of familiar problem domain models, and to have reliable software solutions to their problems automatically generated, without direct involvement of computer experts.

The economic advantage of this approach comes from the ability to automatically reap the benefits of each quality improvement for all past and future instantiations of the template (if past applications are regenerated). We believe that it will be profitable to explore methods for lifting many known program analysis techniques from the level of individual programs to the level of program generation patterns. This should be explored for a variety of issues that range from certifying absence of references to uninitialized variables, absence of deadlock, and many others, perhaps ultimately to template-based proof of post conditions and program termination for generated programs.

To make this vision practical, many engineering issues must be addressed, including presentation issues, methods for lightweight inference [2] and support for transforming and enhancing complex sets of analysis rules. Other issues include systematic methods for dynamic analysis, testing, and debugging of program generation rules. It is not reasonable to expect progress to occur in an instantaneous quantum leap to perfection. A realistic process is a gradual one, where simple sets of program generation rules are deployed, and gradually tuned, improved, certified, and extended. A key issue is enabling rule enhancement and exception closure extension without invalidating all previous effort on analysis and certification of the previous versions.

The difference between the program generation approach proposed here and current compiler generation tools is the associated static analysis capabilities for the program generation rules. It is possible that in the future, ultra-reliable compilers will be built using techniques derived from those introduced in this paper.

REFERENCES

1. F. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner and P. Pepper, *The Munich Project CIP*. Vol. 2: The Program Transformation System CIP-S, Springer, Berlin, 1987.

2. V. Berzins, Light Weight Inference for Automation Efficiency , Proceedings of the *1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Monterey California, 1999.

3. N. Bjorner, Type Checking Meta Programs , Proceedings of the Workshop on *Modeling Software System Structures in a Fastly Moving Scenario*, Santa Margherita, Italy, 2000.

4. T. Reps, *Generating Language-Based Environments*, Doctoral Dissertation, August 1982.

5. D. Volpano, R. Kieburtz, Software Templates , CS/E 85-011, Department of Computer Science and Engineering, Oregon Graduate Center, 1985.

Integrated Design and Process Technology, IDPT 2000
Printed in the United States of America, June, 20CC

REUSE AND RE-ENGINEERING OF LEGACY SYSTEMS*

Jiang Guo and Luqi*
Research Associate
US National Research Council CA, USA
*Department of Computer Science
US Naval Postgraduate School CA, USA

ABSTRACT

Software reuse is widely considered to be a way to increase the productivity and improve the quality and reliability of new software systems. Identifying, extracting and reengineering software components that implement abstractions within existing systems is a promising cost-effective way to create reusable assets and re-engineer legacy systems. This paper summarizes our experiences with using computer-supported methods to develop a software architecture to support the re-engineering of the Janus Combat Simulation System. In this effort, we have developed an Object-Oriented architecture for the Janus Combat Simulation subsystem, and validated the architecture with an executable prototype. In this paper, we propose methods to facilitate the reuse of the software component of these systems by recovering the behavior of the systems using systematic methods, and illustrate their use in the context of the Janus System.

1. BACKGROUND

Rapid changes in hardware and software technology, combined with rapid changes in requirements, require new methods to enable the efficient evolution of current software systems. A significant portion of these systems are real-time control systems that typically have rigid performance and reliability requirements. The ever increasing need to integrate new requirements into these systems poses a challenging problem for the industry as it strives to respond in a timely, accurate manner. There is a lack of reliable methods to maintain and evolve computer based systems.

Software reengineering is the process of understanding existing software and improving it, for increased or enhanced functionality, better maintainability, configurability, reusability, or other software engineering goals. The process involves recovering existing software artifacts and organizing them as a basis for future evolution of the software system. Software reuse is a popular way to increase productivity and improve the quality and reliability of new software systems. Identifying, extracting and reengineering software components which implement abstractions within existing systems is a promising cost-effective way to create reusable assets and re-engineer legacy systems.

We have explored reuse in the context of a case study that addresses the re-engineering of the Janus System. Janus is a software-based war game that simulates ground battles between up to six adversaries. It is an interactive, closed, stochastic, ground combat simulation that features precise color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of a large number of FORTRAN modules, organized as a flat structure and interconnected with one another via FORTRAN COMMON blocks. This software structure makes modification of Janus very costly and error-prone. There is a need to modernize the Janus software into a maintainable and evolvable system and to take advantage of modern personal computers to make Janus more accessible to the Army. TRAC-Monterey is re-engineering Janus into an object-oriented software system that is written in the C++ programming language and operates on personal computers. Prior to rewriting Janus in C++, the software engineering group at the Naval Postgraduate School was asked to extract the existing functionality through reverse engineering and to produce an object-oriented architecture that supports existing and required enhancements to Janus functionality.

Software systems evolve as modifications are made to fix defects or to enhance functionality. Software that has been involved in the evolutionary process for many years often reaches a state where a decision must be made to impose such major changes to the software that significant re-engineering is required. This decision is typically based on factors such as the state of deterioration of the software, high modification costs resulting from reliance of the software on outdated paradigms, ineffective documentation, and obsolescence of hardware platforms on which the software is housed. We have been developing software evolution techniques for several years, and have applied them to the Janus software, which has many of the features listed above.

* This research was supported by ARO(38690-MA), ARO(35037-MA) and DARPA(99-F759).

The complexities associated with the re-engineering of large complex systems and the non-availability of effective conventional methods to address the complexities suggest the need to explore new research directions. One of the historically problematic features of conventional methods is that the models that are produced are typically not applicable across multiple phases of the software development or the software re-engineering process. The software engineer experiences both a syntactic and semantic disconnect from one life-cycle phase to the next. Another problematic feature is that current methods are not sufficiently automatable to feasibly support the re-engineering of complex systems due to lack of effectively computable and accurate methods for extracting and assessing the information that must be analyzed. This research focuses on enhancing software evolution by defining a formal framework which includes methods and representations that are integrable across multiple phases of the software evolution process.

The objectives of this paper are to:

- Describe a formal framework for design recovery. Design recovery is a vital aspect of the software evolution process. We define a formal framework for recovering design information that facilitates the derivation of multiple higher level abstractions with varying levels of formality.
- Explore the reuse and reengineering method of the legacy systems. The method will help to reuse the algorithm and data information extracted from the legacy system and reengineering the system and class structure through re-organizing the data and functions.
- Investigate specification representations. System requirements expressed with formal mathematical representations improve the reliability and maintainability of a system and extend the opportunities for computer aid. We define a methodology that facilitates the creation of specifications of requirements from code.
- Report on our experiences in applying these concepts to the re-engineering of the Janus system.

2. OBJECT ORIENTED MODEL

We are developing a methodology that establishes a formal foundation from which to reengineer systems. The methodology consists of two major steps: the derivation of object-oriented design models and the derivation of formal specifications from the design models.

1) Object-oriented design models: An object-oriented view of a non object-oriented system provides understanding about the behavior and relationships in the system and facilitates the re-engineering of a system to an object-oriented implementation [1]. Object-orientation is the amalgam of three concepts: encapsulation, polymorphism, and inheritance. Encapsulation is realized as a class. Classes are instantiated to create objects, which form the basic run-time entity. Polymorphism refers to the ability of objects to change type during program execution, so that generalized algorithms can be applied to many types of objects. Inheritance defines a relation between classes whereby the definition of a class is based on extending and specializing the definitions of existing classes. It encourages the reuse of classes that are similar by allowing the tailoring of parent classes to meet the needs of a class with similar requirements in a way that meet the requirements of the parent classes. Thus, "inheritance coupled with polymorphism and dynamic binding minimizes the amount of existing code that must be changed when extending a system". We have developed new techniques to derive object representations from non object-oriented code [2].

2) Specifications: We have developed a set of high-level specification tools (CAPS) that formally represent the functionality of legacy systems in an executable form that supports prototyping. A formal specification of a system, which is a description of a system using a notation with a precisely defined semantics, provides clear and precise communication of the system requirements by avoiding the ambiguities of natural language, and thereby reducing design errors and testing time. Benefits of CAPS methods are discussed at length in [3, 4]. A process which includes the creation of a graphic representation of a legacy system from code will serve to not only provide structure and accurate documentation for the system but will also allow the system to utilize the power of graphic specifications for the re-engineering process. We have derived methods to express the functionality of the legacy systems using graphic methods.

The research was motivated by the need for better techniques for the extraction and utilization of desirable functionality of an existing system for re-engineering, reuse, and maintenance. The work was also motivated by the recognition that graphic specifications are currently being used successfully on a broad range of applications in industry because of their potential to decrease software costs and enhance software reliability by helping detect errors. The abstractions will provide suitable representations from which to forward engineer a system and will facilitate the integration of existing requirements with new requirements.

3. REUSING AND RE-ENGINEERING METHODS

We present a new program slicing process for identifying and extracting code fragments implementing functional abstractions. The process is driven by the specification of the function to be isolated, given in terms of a precondition and a postcondition. Symbolic execution techniques are used to abstract the preconditions for the execution of program statements and predicates. The recovered conditions are then compared with the precondition and the postcondition of the functional abstraction. The statements whose preconditions are equivalent to the pre and postconditions of the specification are candidates to be the entry and exit points of the slice implementing the abstraction. Once the slicing criterion has been identified the slice is isolated using algorithms based on dependence graphs. The process has not been specialised for programs written in the FORTRAN or C language. Both symbolic execution and program slicing are performed by exploiting the Data Flow Graph (DFG) and Control Flow Graph (CFG), a fine-grained dependence based program representation that can be used for most software maintenance tasks. The work described in this paper is aiming to explore reverse engineering and reengineering techniques for reusing software components from existing systems.

3.1. PROGRAM SLICING AND INFORMATION EXTRACTING

We extracted dependency and control information to enable the definition of object models. This phase groups together the activities of source code analysis and produces sets of software components. Each one of these sets is a candidate to make up a reusable module when suitably de-coupled, reengineered and possibly generalised. This work includes code structuring, code segmentation, dependency analysis, and finally aggregation to produce design abstractions.

We initiated the design recovery process with a preprocessing step that restructures code. We built on the theory that unstructured code can be written using only D-structures [5] and relied on existing algorithms for that purpose [6]. Our research within this phase involves the use of program slicing techniques for isolating code fragments implementing functional abstractions. Program slicing has been used both as structural and specification driven method. As structural method, program slicing has been used to identify external user functionalities in large programs. The isolation of an internal domain dependent function can be driven by its formal specification. The specification can be used together with symbolic execution techniques to identify a suitable slicing criterion. Code segmentation is needed in order to reduce the granularity and thus the complexity of the remaining processes. We have defined a segmentation scheme that separates the code into modular units while also removing syntactic sugar features of the code. We have also defined heuristics to attach in-code documentation to the appropriate segment. For a program P the result is a set of segments, such that $SG = \{sg_1, sg_2, \ldots sg_n\}$ and $P_f = \cup sg_i$, where $1 < i < n$ and P_f represents code that is identical in functionality to P.

Following the segmentation, we defined dependency algorithms that analyze each sg_i. Specific slicing algorithms that are modified forms of the slicing algorithms found in [7] are employed at the statement, construct, and block levels. These algorithms provide information on all variables: local variables, non-local variables, array variables, and data typing.

The results of the restructuring, segmentation, and dependency steps are segment design representations and a global design representation. These representations include traditional methods, such as call graphs, structure charts, and hierarchical diagrams and other less conventional representations such as variable usage and state change descriptions. These representations serve as input to perform object identification and to create formal specifications of object behavior.

Results of our work include methods that recover the design information at varying levels of granularity, expressible in numerous forms from both data and functional viewpoints. The data and control dependency representations are the basis for our object extraction research.

3.2. REUSABLE COMPONENT CONSTRUCTING

This phase groups together the activities of the analysis of the bag of software reusable component sets singled out in the Program Slicing and Information Extracting phase and produces a set of reusable modules, using reengineering techniques. Also, this phase groups together the activities that produce the specifications of each one of the reusable modules obtained in this phase. Both the functional and the interface specifications must be produced in this phase. We used object-oriented and prototyping techniques to abstract a formal specification from source code modules implementing functional abstractions. Finally, we need to classify the reusable modules and related specifications according to a reference taxonomy. The aim is to re-engineer legacy systems with the reusable modules produced.

Program comprehension is the most expensive activity of software maintenance. The different phases of a reuse reengineering process involves comprehension activities for understanding the structure of existing systems, the functionality implemented by a reuse candidate module and the reengineering effort. We present a method for reuse reengineering existing FORTRAN or C systems. Our goal is to create reusable software components with object-oriented methods.

The problem of extracting encapsulated reusable software components from legacy systems is an area of active research. The concept of the object module as a means of restructuring FORTRAN code into an object-oriented style was introduced in [8]. While code structured as object modules is not truly object-oriented, it marked the beginning of progress along that path. The problem of object identification has been approached by first developing a formal specification of the code and then identifying objects from the formal specification in some methods [9]. In an informal approach, Sward translates code to natural language descriptions and then applies object-oriented analysis and design techniques, such as OMT, to create the object design [10]. A design recovery approach which automatically extracts task flow information utilizing both source code and non-source code information is found in Holtzblatt's work [11]. Other research that addresses behavior abstraction includes object extraction and translation to C++ using data flow analysis [12], partial evaluation for code comprehension [13], and development of new Ada programs by reusing FORTRAN code [14].

In other related work, a complete translation to an intermediate form in the UNIFORM language is used in Lano's work [15] as a bridge to a functional description language and then finally to a Z specification. In some methods, COBOL code is reverse engineered to Z++ and then reengineered to COBOL code using refinement as a part of the REDO (Re-engineering, Documentation, and Validation of Systems) project [16]. A transformation process that creates C++ code from COBOL code is given in [17]. Other work on reverse engineering of COBOL systems to SSADM specifications is a part of the RECAST (Reverse Engineering into CASE Technology) method in which information extracted from source code is represented in PSL to eventually produce input for the physical design phase of SSADM [18]. In an approach that requires a large set of transformations, Ward translates assembler code to a wide-spectrum language (WSL) which contains primitive statements, such as assertions and guards; compound statements, including sequential composition, choice and recursive procedures; and other language extensions including a command language, loops with multiple exits, and mutually recursive procedures [19]. In some approaches, code semantics are expressed as logic specifications [20].

Research that involves the extraction of modules and reusable components from legacy code includes algorithms that construct a hierarchical structure from an implementation description [21], methods to identify abstract data types based on user defined data types [22], direct slicing to extract specific types of code segments [23], identification of cliches to recover program design [24], program segmentation based on focusing and factoring operations on COBOL code [25, 26], and component identification based on formal parameter types and global variables [27]. Methods to abstract the behavior of programs by deriving mathematical expressions from prime programs are found in Hausler's work [28]. An enabling technology which represents software in the form of annotated abstract syntax trees in a persistent object-oriented database and then uses an executable specification language for analysis is described in Markosian's work [29].

We use an incremental approach based on graph-theoretic and set-theoretic concepts. We have investigated reusable component constructed from procedural code to produce intermediate representations from functional and data viewpoints. We then use the intermediate representation to define a high-level object view of the legacy system. Our code and concept abstraction methods include the identification of candidate objects along with their associated attributes and methods.

Our object extraction algorithms are based on the following object model for object O:

$$O = <A, MD>$$

$$A = \{A_1, A_{2,} \ldots\ldots, A_n\}$$

$$MD = \{MD_1, MD_{2,} \ldots\ldots, MD_m\}$$

where A represents attributes and MD represents operations that act on members of A. Our approach is both data-driven and bottom-up. The granularity of a program is viewed at the program, subroutine, and statement levels; however, the primary focus for the unit of functionality is the subroutine. Using the parameters necessary for the execution of each subroutine, the goal is to find the smallest set of parameters needed to obtain the strongest cohesive unit, which becomes a candidate set of attributes for an object type.

We use a greedy approach to the derivation of the A component of O which considers both actual parameters and global variables. To partition the set of actual parameters, AP, where

$$AP = \{AP_1, AP_{2,} \ldots\ldots, AP_n\}$$

a graph-theoretic approach is used. We define an undirected graph G with nodes AP_i, $1 \leq i \leq n$ and with edges connecting AP_i and AP_j if the two parameters both occur in at least one subroutine call. A weight function, W, is then defined to give values to the edges of G. W is computed for all pairs of parameters, AP_i, $AP_j \in AP$, with respect to each subroutine invocation. A constant is used to indicate positive, negative or null contribution to cohesiveness. We define a weighted adjacency matrix M where the value of each M(i, j) is the cumulative value of $W(AP_i, AP_j)$ over all subroutine invocations. Thus, M(i, j) represents a measure of the degree to which parameters AP_i and AP_j are functionally related.

Following the derivation of the weighted adjacency matrix, an initial set of object attributes is determined by using a threshold approach. The potential threshold values are the non-negative real numbers r, such that $r \in M$. For each r, the transitive closure is computed to obtain the attribute sets that are related at that threshold level. The objective is to select the threshold level that produces the largest data sets with the strongest cohesion. Domain knowledge used by a design engineer is encouraged for the selection of the optimal threshold level.

Building on the actual parameter analysis, a similar approach for determining strength among global variables is used. Issues related to the global variables, including aliasing, were resolved. After the determination of the attributes, the method component for an object is determined. We use a state change approach to attach methods to objects. In order to derive the state change information needed, we modified the concept of program slicing from its original definition in Weiser's paper [7]. We perform slicing for each attribute set on a subset of the subroutines and the resultant set becomes a method in the corresponding object.

The result of applying these algorithms is a set of candidate objects. Class abstractions need to be defined over this set to take advantage of the abstraction and inheritance features of object orientation. We have only begun to investigate the class abstraction process. Enhancement of the class abstraction methods is a part of our ongoing work.

3.3. JANUS (A) CASE STUDY

The objective of the case study was to re-engineer an object-oriented architecture for the Janus(A) legacy system. The first step in our process, system and requirements understanding, took the form of a series of brief meetings with the client, TRAC-Monterey, which also included a short demonstration of the current software system. We asked questions and made notes on the system's operation and its current functionality. We paid particular attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. Additionally we collected copies of the Janus User's Tutorial manual, Janus User Manual, the Software Design Manual from a previous version of Janus (3.X/UNIX), and the Janus Version 6.88 Release Notes. Our goal was to gather as much information as we could about the currently existing system to aid in gaining a clearer understanding of its present functionality. The intent of this procedure was to ensure that the system's current functionality was not lost nor misrepresented in the transformation into a more abstract, modular format, and to identify aspects of current system functionality that did not match user needs.

The focus of the re-engineering effort was to abstractly capture the system's functionality and then produce system models that would most accurately represent that functionality, while factoring out independent concerns and aspects that were likely to change.

Armed with the Janus source code, we proceeded to divide the code by directories amongst the team members. Each team member was assigned roughly six to seven directories to explore, examine and gather information. Using manual techniques supported by UNIX commands and review procedures, we were able to get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual to aid in understanding each subroutine's intended function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs.

Using that knowledge, we developed functional models from the data flows. We used an automated tool known as CAPS [3], Computer-Aided Prototyping Systems, version 2.0, developed by Professor Luqi and the Software Engineering group at the Naval Postgraduate School, to assist in developing the abstract models. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to develop our diagrams separately with the associated information flows and stream definitions, and then join them together under the CAPS environment, where they can be used to generate an executable model of the architecture.

Next, we proceeded to develop object models of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. This was probably the most difficult and most important step. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this step, we used our knowledge of object-oriented analysis and applied the OMT techniques and the UML notations to create the classes and associated attributes and operations. This was a crucial step because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software. We used the HP-UNIX systems at the TRAC-Monterey facility to run the Janus simulation software to aid in verifying and/or supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

During this phase of the project, the re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core elements and the object-oriented architecture for the Janus System. They presented the findings to the Janus domain experts from TRAC-Monterey and Rolands & Associates at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center. Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the

Janus System. This revision required creative human effort, as described next.

We used our approach to reuse the information extracted from the old system. The most important type of reuse was reuse of implicit domain models. We reused the domain analysis and knowledge since the domain was stable across the re-engineering transformations. This greatly reduced the time and effort that needed be spent on domain related work, such as the analysis of the domain dependent functions. Second was reuse of implementation concepts. This kind of reuse included the user functionalities, functional abstraction, task flow, and user interface specifications. Third was the reuse of data models. The reuse of data models was very helpful to re-organize the data information although we needed to transform the old data structures into new data structures. Fourth was the reuse of algorithms. The code could not be reused directly because it had to be transformed into another language (Ada). However, the main algorithms were the same – we did not need to redesign the algorithms, we just rewrote them in new languages.

The new architecture of Janus uses an explicit priority queue of event objects to schedule the simulation events. Each event object has an associated simulation object, which is the target of the event. There are 14 event groups, which correspond to the 14 event subclasses. An object oriented approach enabled us to reduce the number of event types needed in the simulation, compared to the legacy code. Depending on the subclass to which an event object belongs, the "execute" method will invoke the corresponding event handler of the associated simulation object to handle the event. The simulation object superclass defines the interface of the event handlers for the event groups, and provides an empty body as the default implementation for the event handlers. The methods are overridden by the actual event handler code at the subclasses that have non-empty actions associated with the events.

This approach enables the same code to handle all kinds of events, including those for future extensions that are yet to be designed. Event objects are created and inserted into the event queue either by the initialization procedure at the beginning of the simulation, by the constructors of simulation objects, or by the actions of other event handlers. Depending on the actual implementation of when and how events are inserted into the priority event queue, it may be necessary to allow events to change their priorities while waiting in the queue. The priority of an event is determined by the time at which the event is supposed to occur, and by event type in case more than one event is scheduled at the same time.

One of the objectives of the reengineering effort was to add the capability for a Janus simulation to interact with other simulations in a distributed environment. To accomplish this, World Model object subclasses were created to provide specialized methods for the world modeler to update objects from other simulators. Information concerning objects local to the Janus simulator can be broadcast over the simulation network, either periodically by an active world modeler object, or by individual local objects whenever they update their own states.

3.4. EVALUATION OF RESULTS

We tested our methods for identifying objects on a set of programs ranging from 500 lines of code to 10,000 lines of code. As a part of our test bed, we used programs from the Janus (A) which were developed by DoD. Our test protocol was to begin the testing process with small programs so that the dependency and slicing information could be validated manually. The testing strategy was to choose test programs that exhibit different code characteristics, particularly related to the use of global variables. We were able to manually verify the accuracy of the extraction routines on small systems.

We then applied the methodology to medium-sized programs and evaluated the results. Our evaluation process included the identification of a set of metrics against which to measure the designs. Metrics in the reverse engineering area are sparse. We adopted the approach of measuring our success using the following . three metrics:

M.1 Functional equivalence of newly created and original designs.
M.2 Quality of newly created design.
M.3 Reuse rate of the original program.

M.1 Functional equivalence of newly created and original designs

The design of a program S1 is functionally equivalent to the design of program S2 if when they are executed with identical inputs, they produce identical outputs. This is a critical measure. To assess the functional equivalence of our abstracted designs, we implemented the designs in an object-oriented environment and then ran test cases on the new and the old systems. Based on our test cases, the test systems were functionally equivalent.

M.2 Quality of new designs

Our view of a significant metric is the quality of the resulting design; however, measuring quality is far from straightforward. We based our findings in this area on the traditional view of design quality in terms of modifiability, modularity, levels of abstraction, loose coupling, and high cohesion [30]. We also considered metrics that have been derived specifically for object-oriented designs, including depth of inheritance tree (DIT), number of children (NOC), response for a class

(RFC), and lack of cohesion in methods (LCOM) [31]. Coupling can be measured by DIT and NOC; cohesion can be measured by LCOM; abstraction measured by DIT and NOC; modifiability can be measured by RFC and LCOM; and modularity measured by DIT and NOC.

For our case studies, we found low measures for both DIT and NOC which is expected based on the conservative view of creating the subclasses, medium measure for RFC due to global variable usage, low LCOM because the methodology insures cohesion in the creation of the objects. Thus, the designs were low on coupling, high on cohesion, and generally good on modifiability.

M.3 Reuse rate of orginal program

Reuse rate of the program is measured by the percent of the program that is actually utilized in the extraction process. If reuse rate is not 100%, one of two cases occurs: 1) some of the system functionality may not be preserved, or 2) statements not extracted represent dead code. However, 100% reuse rate does not imply functional equivalence, and vice versa. The reuse rate for our test programs was in all cases greater than 40%. This measure gives another perspective from which to assess the quality of the newly created design abstractions.

4. CONCLUSION

Successful re-engineering requires a delicate balance between creative concepts for requirements enhancement and computer aid. Bottom-up tools can help guide this creative process and help to ensure its accuracy.

Our experience in this case study suggests that prototyping and reuse can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed.

In particular, we found that constructing even a very thin skeletal instance of the proposed new architecture raised many issues and enabled us to correct, complete, and optimize the architecture for both simplicity and performance.

The computer-aided prototyping tools in the CAPS system enabled us to do this with a minimal amount of coding effort. The bulk of the code was generated automatically, enabling us to concentrate on system structuring issues, to consider and evaluate various alternatives, and to improve the design while doing detailed manual implementation for only a few pages of critical code.

REFERENCES

[1] Rivera, R., "Knowledge-Based Metalangauge-Based Object Abstraction for Automatic Program Transformation", *Proceedings of the 4th Systems Re-engineering Technology Workshop,* 1994, pp. 319-326.

[2] Jiang Guo and Luqi, "Object Modeling to Re-engineering Legacy Systems", *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering,* Kaiserslautern, Germany, June 1999, pp. 346-353.

[3] Luqi, "Computer-Aided Prototyping - Status and Experiments", *Proceedings of International Symposium and Workshop on New Models for Software Architecture*, Kanazawa, Japan, Nov. 8, 1993, pp. 23-30.

[4] Luqi, "Software Evolution via Rapid Prototyping", *IEEE Computer,* vol. 22, no. 5, May 1989, pp. 13-25.

[5] Dijsktra, E. W., A Discipline of Programming, Prentice Hall, 1976.

[6] Boehm, C and Jacopii, G., "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules", *Communications of the ACM,* vol. 9, no. 5, May, 1966, pp. 366-371.

[7] Weiser, M., "Program Slicing", *IEEE Transactions on Software Engineering,* vol. SE-b, No.4, July, 1984, pp. 352-357.

[8] Zimmer, J. A., "Restructuring for Style", Software Practice and Experience, vol. 20, no. 4, 1990, pp.*365-389.*

[9] Gannod, G. C. and Cheng, B. H. C., "A Two-Phase Approach to Reverse Engineering Using Formal Methods", *Proceedings of Formal Methods in Programming and Applications Conference*, June, 1993, pp. 335-348.

[10] Sward, R. E., and Steigerwald, R. A., "Issues in Reengincering from Procedural to Object-Oriented Code", *Proc. 4th Systems Re-engineering Technology Workshop",* 1994, pp. 327-333.

[11] Holtzblatt, L. J., Piazza, R. L., Reubenstein, H. B., Roberts, S., and Harris, D. R., "Design Recovery for Distributed Systems", *IEEE Transactions on Software Engineering,* vol. 23, no. 7, July, 1997, pp. 461 - 472.

[12] Ong, C. L. and Tsai, W. T., "Class and Object Extraction from Imperative Code", *Journal of Object-Oriented Programming,* April, 1993, pp. 58-68.

[13] Blazy, Snadrine, and Facon, P., "Partial Evaluation as an Aid to the Comprehension of FORTRAN Programs", *Proceedings of 2nd Workshop on Program Comprehension,* 1993, pp. *46-54.*

[14] Wilkening D, Loayll, E., Pitarys, M, and Littlejohn, K., "A Reuse Approach to Computer-Assisted Software Re-engineering," *Proceedings of the Systems Re-engineering Technology Workshop,* 1994, pp. 83-90.

[15] Lano, K. and Haughton, H., "Integrating Formal and Structured Methods in Reverse Engineering", *Proceedings of the Working Conference on Reverse Engineering,* Baltimore, MD. May, 1993, pp. 17-26.

[16] Bowen, J. P., and Hanchey, Michael G., "Ten Commandments of Formal Methods", *IEEE Computer,* April 1995, pp. *56-63.*

[17] Leite, J.C., Sant, M., and Prado, A., "Porting COBOL Programs Using a Transformational Approach", *Journal of Software Maintenance,* vol. 9, no. 1, 1997, pp. 3-30.

[18] Edwards, H. and Munro, M., "RECAST: Reverse Engineering from COBOL to SSADM Specifications", *Proceedings of 5th Int. Conference on Software Engineering,* pp. 499-508.

[19] Ward, M. P., and Bennett, K. H., "A Practical Program Transformation System for Reverse Engineering", *Proceedings of Working Conference on Reverse Engineering,* 1992, pp. 212-221.

[20] Gannod, G. C. and Cheng, B.H.C, "Using Informal and Formal Techniques for the Reverse Engineering of C Programs", *Proc. 3rd Working Conference on Reverse Engineering,* 1996, pp. 249-258.

[21] Choi, S. C. and Scacchi, W., "Extracting and Restructuring the Design of Large Systems", *IEEE Software,* January, 1990, pp. 66-73.

[22] Canfora, G., Citile, A. and Munro, M., "A Reverse Engineering Method for Identifying Reusable Abstract Data Types", *Proceedings of the Working Conference on Reverse Engineering,* May, 1993, pp. 73-82.

[23] Cutillo, F., Fiore, P and Visaggio, G., "Identification and Extraction of Domain Independent Components in Large Programs", *Proceedings of Working Conference on Reverse Engineering,* 1993, pp. 83-92.

[24] Rich, C. and Wills, L., "Recognizing a Program's Design: A Graph-Parsing Approach", *IEEE Software,* Jan. 1990, pp. 82-90.

[25] Ning, J. *Q.*, Engberts, A. and Kozaczynski, W., Recovering Reusable Components from Legacy Systgems by Program Segmentation", *Proceedings of Working Conference on Reverse Engineering,* 1993, pp. 64-72.

[26] Ning J. *Q.*, Engberts, A., and Kozaczynski, W., "Automated Support for Legacy Code Understanding", *Communications of the ACM,* vol. 37, no. *5,* May, 1994, pp. *50-57.*

[27] Liu, Sying-Syang and Wilde, N., "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery", *Proc. 1990 Conference on Software Maintenance,* pp. 266-271.

[28] Hausler, Phillip A. and Pleszkock, Mark G., "Using Function Abstraction to Understand Program Behavior", *IEEE Software,* Jan. 1990, pp. *55-64.*

[29] Markosian, Lawrence, Newcomb, P. Brand, R., Burson, S. and Kitzmiller, T., "Using an Enabling Technology to Reengineer Legacy Systems", *Communications of the ACM,* vol. 37, no. *5,* May, 1994, pp. *58*-70.

[30] Achee, B. L. and Carver, D. L., "Creating Object-Oriented Designs From Legacy Code", *Journal of Systems and Software,* February 1997, pp. 30-41.

[31] Chidamber, S. R., and Kemerer, C. F., "A Metrics Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering,* vol. 20, no. 6, June, 1994.

A Survey of Software Reuse Repositories*

Jiang Guo
Research Associate
US National Research Council
NPS/CS, Monterey, CA 93943, USA
Gj@cs.nps.navy.mil

Luqi
Department of Computer Science
US Naval Postgraduate School
Monterey, CA 93943, USA
Luqi@cs.nps.navy.mil

Abstract

Reuse libraries are organizations of personnel, procedures, tools, and software components directed toward facilitating software component reuse to meet specific cost-effectiveness and productivity goals. The paper gives a survey of the major software reusable component repositories. This survey will be a base to develop future efficiently searchable, user-friendly, useful, and well-organized repositories.

1. Introduction

Reuse libraries are directed toward facilitating software life cycle component reuse to meet cost-effectiveness and productivity goals [1]. The principal rationale for the existence of a reuse library is to provide ready access to reusable components by the staff of development and maintenance organizations, and to support system composition and rapid prototyping [2, 3]. The number of cases in which library systems are successfully being used to maintain code and other reusable software life cycle components continues to increase. It is essential that the library system support developers and other users in the process of locating, retrieving, comparing, and maintaining reusable software components.

Reuse libraries are only one critical element of successful reuse program. In the past, reuse has primarily been the result of opportunistic success, where one program was able to take advantage of the efforts of another. There must be a paradigm shift from current software engineering and development practices to a software engineering process in which software reuse is institutionalized and becomes an inseparable part of the software development process. Reuse must be systematic, driven by a demand for software components identified as a result of domain analysis and architecture development. Reuse needs to be treated as an integral part of engineering and acquisition activities. Most importantly, it is essential that an organizational infrastructure be implemented to manage domains, define products and standards, establish ownership criteria, allocate investment resources, and direct the establishment and population of reuse libraries. An effective infrastructure will guide reuse activities to avoid duplication of effort, impose necessary standardization, and ensure library population is user demand-driven.

2. Library Mechanism

Usually, critical reuse library capabilities include the following:

- automated library system with a Graphical User Interface, for browsing, searching and retrieval;
- standard component framework (e.g., to include purpose, functional description, certification level, key environmental constraints, historical results of usage and legal restrictions);
- effective classification scheme for each domain; and,
- thorough system and component documentation.

Each library system must be designed to provide as much automated support as possible to users in identification, comparison, evaluation, and retrieval of similar reusable components. Support for adapting, transforming, and specializing components is desirable. It must also provide a range of support to users in locating and comparing the relative reusability of individual library components. Furthermore, the system must be readily available to system developers if it is to be used, and must support access from a variety of platforms. As the library acquires significant number of Reusable Software Components (RSCs), an automated search and retrieval system becomes indispensable [4, 5, 6]. Whatever tool is used, the library must have a way to classify RSCs so that a user can quickly find what is wanted without frustration and delay. Sophisticated, expert system, knowledge-based approaches and new technologies for high-speed text search are the subjects of current research efforts. Generally speaking, software reusable component retrieval

* This research was supported by ARO(38690-MA) and DARPA(99-F759).

methods include browsing, keyword searching, facet approach, syntactic matching, and semantic matching [1].

Standard component frameworks help ease the process of comprehension and comparison of similar components, and include data such as relative numeric measures for reusability, reliability, maintainability and portability [7]. Inclusion of testing and component documentation provides additional information to help the potential user gauge the effort required to tailor the component for reuse.

Effective classification schemes are essential to assist the user in locating and comparing library components, and to speed the process of identifying appropriate components for the task at hand [8, 9]. Finally, system and component documentation complete the cycle of evaluation, and enable the reuser to determine which components have reuse potential with regard to specific requirements, and to fully comprehend the process of obtaining components for reuse in a new application.

In addition, other equally important requirements have been identified that require resolution in order to support cohesive, wide reuse. These include l) integration of library capabilities and procedures within the system development and acquisition process; 2) identification and support of specific requirements associated with the security and integrity of reusable components implementing Trusted Computing Base (TCB) or other security capabilities; and 3) intercommunication and interoperabllity among diverse library systems. Experience has shown that these requirements can only be resolved through the combination of developing technology, standard procedures and evolution or revision of existing policies.

There are different communities for which a repository is necessary, and each community has somewhat different repository requirements. These communities include the national or horizontal communities; the local or, internal communities, and a number of domain-specific vertical communities [10].

3. Library Operation

The reuse library, while essential, is but one ingredient in a successful reuse program. Experience has shown that actual support of reuse activities within a target domain must include a range of programmatic and technological support that includes domain analysis activities, user indoctrination and training, metrics collection and analysis, reuse engineering support, and component certification and reengineering.

The importance of domain analysis activities as an initial step in implementation of a reuse library cannot be over-emphasized. Domain analysis activities are considered to be an integral part of providing reuse support to various programs. Standard products of domain analysis include identification of high-demand categories of reusable components, domain-specific models and architectures, and domain specifications and taxonomies. These direct products also provide the basis for development of long-term implementation plans and domain knowledge bases.

In order to measure reuse success, the library must collect and analyze considerable data in a continuing assessment of the library's procedures and tools, the usefulness of its RSC collection, the accuracy of RSC classifications, and the general responsiveness of the library to the needs of users.

The library staff receives direction in the form of specific operational objectives, principally aimed at making software reuse cost-effective. In addition to ensuring that RSCs are available, the library is in a position to provide other support to help ensure that the benefits of reuse are realized, including the distribution of published manuals like Standards and Guidelines and user documentation for library tools. In addition, on-call assistance should be made available to users. Reuse engineering support encompasses a wide range of engineering activity. These activities will include working within individual system development and maintenance efforts to assist in (l) identification, selection and reapplicatlon of existing reusable software components, (2) quantification of potential savings or cost avoidance as a result of reuse, and (3) design and implementation of software products that will themselves be reusable in future efforts.

Another key area is thorough library system documents. Documentation has proven to be an essential aspect in establishing and operating a library.

4. Some Reusable Software Component Repositories

4.1. Commercial Repositories

- **+1Reuse Repository**

The +1Reuse system was developed by +1 Software Engineering Co. in California [31]. It is now running on Sun Workstation platforms. Operating system is Solaris. GUI is based on OpenWindows, Motif, and CDE.

The +1Reuse system supports reuse repositories created and maintained by the user, project-wide "filtered" repositories under strict quality controls, and selective reuse. Selective reuse enables reuse of any submodel from an existing or re-engineered +1Environment project. In a sense, every +1Environment project is a reuse library. Selective reuse significantly improves a user's ability to reuse all source code and documentation from all previous projects and at any granularity. (To the best of our knowledge, they are the only company to support this feature.)

The +1Reuse system supports reuse of: design, documentation, source code, header files, test cases, test shell scripts, expected test results, and modeling information.

All source code reversed engineered or developed using the +1Environment can be reused. +1Reuse addresses reuse issues such as reuse of source code under configuration management and duplicate file names. +1Reuse supports three forms of reuse: User-Defined Reuse Library, Filtered Reuse Library, and Selective Reuse. Since a programmer's productivity can be increased by reusing existing code and documentation, +1Reuse helps to make all source code, documentation, header files, and test files reusable by its support of submodels. After a submodel has been selected, +1Reuse copies the submodel and its associated files to the new project and helps to resolve a number of problems which may arise (e.g., identical file names and files checked in under configuration management).

- **Software Asset Library Management System (SALMS)**

SALMS is a system for classifying, describing, and querying reusable assets [32]. Reuse of software assets at all phases of the software engineering life-cycle is recognized as being one of the major enablers for productivity and quality improvements. However, a common inhibitor to company-wide reuse is often the lack of visibility of reusable assets within the developer community.

A central repository for reusable assets provides a solution to this problem. The main purpose of such repository is to provide mechanisms for classification and storage of software assets, along with techniques for efficiently retrieving them.

SALMS (Software Asset Library Management System) is a software product which provide these mechanisms. It fills the gap between development for-reuse activities (building, acquiring, or re-engineering of reusable assets) and the development with-reuse activities (using reusable assets in the creation of new software products). It plays a central role in the implementation of a company's reuse program.

In addition, SALMS also provides features for the requirement management activity, and for the creation and management of a company's technical library. SALMS can be distributed over customer's network of PCs or UNIX workstations and thus be accessible by all developers within a software organization. The user interface is based on WEB Technology.

In SALMS, an asset can be viewed as a collection of artifacts produced throughout the life-cycle, such as requirements, architecture models, design specifications, source code, or test scripts.

- **Automated Software Reuse Repository (ASRR)**

The Automated Software Reuse Repository (ASRR) tool provides users with a searchable repository of reuse information [33]. It consists of two main parts, the administration tool and the reuse repository. The administration portion of the tool performs user administrative functionality including: the ability to add, delete, or change users and their attributes. The attributes include the following: security levels, group and security permissions to add, edit and delete modules. The reuse repository allows the user to upload modules and store them in a searchable repository.

The ASRR provides the following functions:

- Program Control. Provides complete login control for the ASRR.
- Protection. The ASRR can limit a user's edit, delete, viewing, add, upload and download module permissions through the administration portion of the tool.
- Security. The ASRR tool provides extra security for inactive users by logging them out of the ASRR after a 30-minute period of inactivity.
- Easy Access to Reuse Items. The ASRR tool allows registered users flexibility in searching for reuse items in the reuse repository by allowing the users to search for strings of words using "not", "or", or "and" in searching.
- Reuse Information Readily Available for Users. Specific information is available for reuse module items including the platforms utilized, ease of reuse and any additional information obtained from users.

- **The Universal Repository**

The Universal Repository was developed by Unisys [34]. It is designed to help customers move forward into a repository-based development environment.

The Universal Repository, which is based on object-oriented principles, can function as the backbone of a flexible workgroup or enterprise development environment. At the core of this repository is the Repository Services Model (RSM) - which can encompass representations of all tools, database management systems (DBMSs), programming languages, business rules, and data.

Customers can extend the Universal Repository by adding their own models based on the structures provided in the RSM. The summation of all models defined in a repository is called the information model. Each part of customers' development environment becomes an integrated piece of the whole when customers use the models encompassed within the information model. This unified view enables both developers and customers to achieve inter-tool integration.

In addition to its modeling capabilities, the Universal Repository offers features that enhance customers' development environment, manage organizational information, and make such information available to everyone in a customers' organization.

Unisys is dedicated to improving customers' product lines with the Universal Repository. Support and training are available to help customers quickly adopt this new technology. By providing a shared catalog of all software components, a repository promotes reuse. It makes it easy to locate and access components for reuse in multiple applications. Reusing software components can enhance quality. Customers can develop, validate, and verify a component for use in one product. When customers reuse that component, they expend less time and fewer resources to validate and verify that component for use in other products [11]. A single change to correct a defect in a reused component is reflected in all tools using that component. Such consistency among products ensures their integration and interoperability when you port them to different operating platforms.

- **AIRS**

AIRS is an AI-based library system for software reuse, which was developed by E.J. Ostertag, J.A. Hendler, R. Prieto-Diaz, C. Braun [12]. AIRS allows a developer to browse a software library in search of components that best meet some stated requirement. A component is described by a set of (feature,term) pairs. A feature represents a classification criterion, and is defined by a set of related terms [10, 12]. AIRS also allows representation of packages, that is, logical units that group a set of related components. As with components, packages are described in terms of features. Unlike components, a package description includes a set of member components. Candidate reuse components (and packages) are selected from the library based on the degree of similarity between their descriptions and a given target description [13]. Similarity is quantified by a non-negative magnitude (called distance) that represents the expected effort required to obtain the target given a candidate. Distances are computed by functions called comparators. Three such functions are presented: subsumption, closeness, and package comparators. The AIRS classification approach is based on a formalization of the concepts and is similar to faceted classification [44]. The functionality of a prototype implementation of the AIRS system is illustrated by application to two different software libraries: a set of Ada packages for data structure manipulation, and a set of C components for use in Command, Control, and Information Systems.

- **Reuse Library Toolset (RLT)**

EVB Software Engineering, Inc. announced the commercial release of the Reuse Library Toolset (RLT) in 1994 [35]. RLT is a system for creating and managing collections of reusable assets independent of programming language, design method, or development process. To represent all life-cycle assets RLT employs the Extended Faceted Classification System, controlled keyword, attribute value (frames), and asset interdependencies.

Experience has shown that the cost of producing software is significantly reduced when reuse is an integral part of the process. RLT supports all reuse oriented tasks, from library management through domain analysis to asset search and retrieval. With its intuitive graphical user interface, RLT is easy for beginners to learn, yet provides powerful functionality for advanced users with complex needs.

RLT provides reuse and library metrics, client-server architecture, and ability to exchange library information across multiple platforms and databases. These include: DEC Alpha OSF1, HP/UX, SGI, SunOS, Solaris, Informix, Oracle, and Sybase. Additional platforms have been supported in 1995 include: Windows 3.1/NT and OS/2.

RLT's open architecture allows easy integration with existing CASE and development tools, such as structure design tools, versioning systems and configuration management systems.

- **HSTX Reuse Repository**

The HSTX Reuse Repository was developed by Hughes STX Corporation [36]. The mechanisms are designed so users can search/browse the contents of the Reuse Repository for what they need and submit contributions to the Reuse Repository librarian through WWW pages.

4.2. Government Repositories

- **Defense Software Repository System (DSRS)**

The DSRS is an automated repository for storing and retrieving Reusable Software Assets (RSAs) [14]. The DSRS software now manages inventories of reusable assets at seven software reuse support centers (SRSCs). The DSRS serves as a central collection point for quality RSAs, and facilitates software reuse by offering developers the opportunity to match their requirements with existing software products.

DSRS accounts are available for Government employees and contractor personnel currently supporting Government projects. The Account Request Form must be approved and signed by the requestor's Government Project Manager prior to submission to the SRP. The Customer Assistance Office (CAO) is the SRP point of contact for both technical and non-technical information and support.

The Defense Software Repository System (DSRS) supports reusable asset classification to comply with published guidance (DoD 8020.1-M and TAFIM), support domain engineering, establish more effective asset searching, and increase interoperability. The DoD software community is trying to change its software engineering model from its current software cycle to a process-driven, domain-specific, architecture-based, repository-assisted way of constructing software [15]. In this changing environment, the DSRS has the highest potential to become the DoD standard reuse repository because it is the only existing deployed, operational repository with multiple interoperable locations across DoD. Seven DSRS locations support nearly 1,000 users and list nearly 9,000 reusable assets. The DISA DSRS alone lists 3,880 reusable assets and has 400 user accounts.

DSRS is adaptable to additional types of reusable assets and better methods of describing them. The description of repository assets is called classification. This paper reports the results and recommendations of a study of classification methods for storage and retrieval of Reusable Assets (RAS) in the DSRS. The Defense Software Repository System (DSRS) reusable asset classification is changing to achieve policy compliance, support domain engineering, establish more effective asset searching, and increase interoperability.

The far-term strategy of the DSRS is to support a virtual repository. These interconnected repositories will provide the ability to locate and share reusable components across domains and among the services. An effective and evolving DSRS is a central requirement to the success of the DoD software reuse initiative. Evolving DoD repository requirements demand that DISA continue to have an operational DSRS site to support testing in an actual repository operation and to support DoD users. The classification process for the DSRS is a basic technology for providing customer support [16]. This process is the first step in making reusable assets available for implementing the functional and technical migration strategies.

- **Library Interoperability Demonstration (LID)**

The Library Interoperability Demonstration (LID) is a prototype library system [17, 18]. It is used to illustrate how monolithic reuse libraries can be decomposed into distinct, functional layers connected by open interfaces, such as those specified by Asset Library Open Architecture Framework (ALOAF). It is a collaboration between SAIC and Unisys. The demonstration shows how the physical storage of assets can be separated from the cataloging of assets, and how a user can choose a single, local, user interface tool to access multiple reuse libraries.

The STARS Program developed a specification of an ALOAF to support an "open systems" approach to constructing asset libraries. The ALOAF evolved to incorporate interfaces specifically intended for interoperability, culminating in the release of ALOAF Version 1.2 [19]. The LID builds upon the open interfaces provided by ALOAF, its associated Asset Interchange Language (AIL), PCTE, OSF/Motif, and POSIX. As shown in the LID Software Architecture diagram, a reuse library can be divided into three distinct layers which are connected via open interfaces, thus providing opportunities for interoperability at each layer. The three layers are:

- User Interfaces. The demonstration includes two user interface tools: a graphical browser derived from the Unisys Reuse Library Framework (RLF) and a text-based browser modeled after SPS's InQuisiX reuse library system. Both tools are built upon Ada bindings to OSF/Motif.
- Asset Catalogs. The demonstration shows two asset catalogs. The first catalog is derived from the Unisys collection of ASW components, and resides on an IBM RISC System/6000 at the STARS Technology Center. The second catalog is derived from SAIC's collection of flight simulator components, and resides on an IBM RISC System/6000 at the SAIC offices in Orlando, FL. The interface between each of the catalogs and the user interface tools is defined by the ALOAF.
- Asset Storage. In the demonstration, the storage of assets is provided by the AFS cell at the STARS Technology Center. Neither catalog stores assets itself; instead, both catalogs "subcontract" the storage function to the AFS server.

- **Integrated - Computer Aided Software Engineering (I-CASE)**

I-CASE was developed by Air Force Reuse Center (AFRC) [38]. The Air Force Reuse Center is the Air Force Management Information Systems (MIS) repository for reusable software assets. These assets are primarily Ada source code modules consisting of Government and commercial packages. The library has over 1,200 assets including many assets of the system life-cycle, such as requirements, designs, documentation and source code. Integrated Computer-Aided Software Engineering (I-CASE) provides a contract for DoD users to purchase an integrated set of tools that will automate many of the MIS software development activities over the entire software development and maintenance life-cycle. I-CASE also provides the support elements necessary to implement, operate, and maintain the I-CASE environment (i.e., training, maintenance, and technical support). The overall strategy of this project is to automate reuse processes through an Integrated-Computer Aided Software Engineering (I-CASE) environment. The specific strategy is to implement these reuse processes within a workflow environment to certify or re-engineer reusable assets as quickly as possible. The EVB Reuse Library Tool (RLT), supplied as part of I-CASE, is used as the reuse repository tool.

- **Multimedia Oriented Repository Environment (MORE)**

As the World Wide Web (WWW) becomes very popular among internet users, an increasing number of public repositories are using the WWW to promote their services. The Electronic Library Services and Applications (ELSA) project is the operational part of the Repository Based Software Engineering (RBSE) program [20]. RBSE is a National Aeronautics and Space Administration (NASA) sponsored program dedicated to introducing and supporting common, effective approaches to designing, building, and maintaining software systems by using existing software assets stored in a specialized library or repository.

In addition to operating a software lifecycle repository, RBSE promotes software engineering technology transfer, academic and instructional support for reuse programs, the use of common software engineering standards and practices, software reuse technology research, and interoperability between reuse libraries/repositories.

During its life cycle, the ELSA project responded to emerging technologies, the growing sophistication of its client base, and industry trends by advancing the capabilities of its management software. Thus, ELSA stands as a customer-driven environment employing an advanced library management mechanism, MORE (Multimedia Oriented Repository Environment).

ELSA replaced AdaNet on August 31, 1994 when the first public access to its new service was granted. The library is the operational part of the Repository Based Software Engineering (RBSE) program which is a NASA sponsored initiative in software reuse. In a timeframe of approximately two weeks, ELSA transitioned its library holdings and accompanying metadata from a monolithic X-Windows based system to MORE. The improved interface employs client/server technology and is accessible through the WWW. MORE is a public domain, metadata based repository tool employing the WWW as its sole user interface. It consists of a set of application programs which operate together with a stock httpd server to provide access to a database of metadata [21]. The entire interface, client browsing and searching, repository definition, data entry and other administrative functions, are provided through stock Web clients.

Repository assets are classified using a collection (topic) and class (type) paradigm. According to their subject matter, they are included in the collections or subordinate collections that best represent domain coverage. The assets are also classified by media or information type through the class approach. Thus, users can view the information from a top-down perspective through the hierarchy of collections or across collections by the hierarchy of classes.

MORE was designed to support this collection and class model. Navigation is achieved through the activation of high-level hypertext links which ultimately lead to metadata or assets themselves. Searching (Natural Language or Pattern Match) is performed against information provided in the metadata [22, 23, 24]. This combination provides users with a reliable and efficient means of accessing a high volume of assets.

Administrative functions are specifically designed to meet librarians' needs. For instance, assets are stored in "developmental" mode which provides a cleanroom environment for the performance of population and/or certification activities. Developmental assets are only available for viewing by librarians. Following the completion of these processes, each asset is promoted to "production" mode and is therefore accessible to the general user population.

Each collection can have one or more groups associated with it that are authorized to access the assets and subcollections making up the collection. Groups in turn are made up of sets of users and other groups; all defined through the librarian interface. Users not transitively a member of a designated group for a given collection will never see the collection, or its contents, through any of the browser or search mechanisms. This mechanism supports the definition of multiple virtual repositories in a single physical repository, reducing administrative overhead and allowing direct sharing of assets.

- **Asset Source for Software Engineering Technology (SAIC/ASSET)**

Asset Source for Software Engineering Technology (SAIC/ASSET) offers products and services in digital library support, electronic commerce and software engineering with an emphasis on reengineering and reuse [26]. SAIC/ASSET, established by Advanced Research Projects Agency (ARPA) as a subtask under the Software Technology for Reliable Systems (STARS) program, is transitioning to a private enterprise as a division of Science Applications International Corporation (SAIC).

SAIC/ASSET's primary mission is to provide a distributed support system for software reuse with the Department of Defense (DoD) and to help foster a software reuse industry within the United States. SAIC/ASSET's initial and current focus is on software development tools, reusable components and documents on software development methods. SAIC/ASSET is participating in interoperation with other reuse libraries such as:

- Comprehensive Approach for Reusable Defense Software (CARDS)
- Ada and Software Reuse Information Clearinghouse Defense Software Repository System (DSRS)
- Electronic Library Services & Applications Lobby (ELSA)

The goals SAIC/ASSET are pursuing involve:

- Creating a focal point for software reuse information exchange
- Advancing the technology of software reuse
- Providing an electronic marketplace for reusable software products to the evolving national software reuse industry.

To achieve these goals, SAIC/ASSET operates the Worldwide Software Reuse Discovery (WSRD) Library. The WSRD Library is populated with quality reusable software components which can be distributed to its subscribers. WSRD contains over 700 assets available to over 1500 users throughout the world. The library specializes in software lifecycle artifacts and documents written specifically to promote software reuse and development. SAIC/ASSET users have access to other components stored in the CARDS and DSRS reuse libraries. Through the WSRD, users can search, browse and download asset catalogs in over 30 domains. SAIC/ASSET's World Wide Web pages, located at http://source.asset.com/, describe products and services offered through SAIC/ASSET, as well as information related to software reuse.

- **The Public Ada Library (PAL)**

Since 1984, the Ada Software Repository (ASR) has been a major, publicly available source of Ada code. Now called the Public Ada Library (PAL) [27], it provides more than 100 megabytes of programs, components, tools, general information, and educational materials on Ada. It also contains materials on the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), which is based on Ada.

For those with access to the Internet, the PAL can be accessed via the File Transfer Protocol (FTP). The PAL is located on the wuarchive.wustl.edu host, and on mirror sites at ftp.cnam.fr and ftp.cdrom.com. Also, the PAL can be obtained on disk, tape, and compact-disk read-only memory (CD-ROM).

Additionally, the PAL can be accessed by means of such Internet services as: the Network File System (NFS), which allows computers to share files across a network; archive, a system of querying anonymous-FTP sites; and gopher, via gopher servers wuarchive.wustl.edu and gopher.wustl.edu.

- **CAPS Software Reusable Component Repository**

CAPS (Computer Aided Prototyping System) is a research project developed by the Software Engineering Group led by Prof. Luqi at Naval Postgraduate School [39]. Initial implementation of CAPS software base was first explored in 1988 [40]. An implementation of the software base was accomplished in 1991 by using ONTOS, an object oriented data base management system that provides an interface to C++ for customization and flexibility [41]. The CAPS software base is being changed to a software component repository since 1998 [1]. The CAPS component repository supports two critical functions, component storage and component retrieval. Much effort has been made to improve the component retrieval method [42, 43]. To the best of our knowledge, CAPS Repository is the only one that supports profile matching and signature matching. It provides high precision and recall retrieval method at same time [1]. The CAPS repository is still under construction. A prototype has been developed to verify the performance of the retrieval methods [1].

- **The Ada Library and the Reuse Library at the Defense Information Systems Agency (DISA)**

The Ada Library and the Reuse Library at the Defense Information Systems Agency (DISA) are public, non-lending, reference libraries for all professionals, students, and researchers seeking information on the Ada programming language and on software reuse [37]. The number of books and articles on Ada and on reuse grows daily. Also, there is a wealth of information available on the Internet and on the World Wide Web. Putting the Net together with the Ada and Reuse Libraries makes a very powerful research tool.

Both Libraries collect and hold information found in documents, books, conference proceedings, newspaper and journal articles, and other multimedia material.

The Libraries can provide assistance in two ways: helping users find publications in each library, and conducting on-line searches for published information available elsewhere. Users can access these resources in person, and via the Web, or they can call DISA to request a search.

Over the Web, go to http://sw-eng.falls-church.va.us. There, click on "Library" at the main page of either the AdaIC or the ReuseIC. Users can search database by title, author, subject, or publisher.

5. Comparison

Commercial reusable component repositories usually are integrated into a CASE environment [28, 29]. Currently, some major repositories (ASSET, PAL, and DSRS) begin to use web-based techniques to provide services. They are utilizing flat files written in HyperText Markup Language (HTML). Electronic Library Services and Applications (ELSA) has gone a step further by using the Multimedia Oriented Repository Environment (MORE).

Following are some comparison results for all the repositories listed above.

Features	Web-Based	Integrated into CASE Environment	Security Control	Retrieval Methods
+1 Reuse Repository		Y	Y	Browsing
SALMS	Y	Y	Y	Keywords
ASRR			Y	Keywords
The Universal Repository		Y	Y	Browsing and Keywords
AIRS			Y	Facets Approach
RLT			Y	Keywords
HSTX Reuse Repository	Y		Y	Keywords
DSRS	Y		Y	Keywords
LID			Y	Keywords
I-CASE		Y	Y	Keywords
MORE	Y		Y	Keywords
ASSET	Y	Y	Y	Keywords
PAL	Y		Y	Keywords
CAPS		Y		Browsing, Keywords, Profile & Signature Matching
Ada Library and Reuse Library (DISA)	Y		Y	Browsing and Keywords

6. Conclusion

Web-based reuse is the trend of software component repositories supported by the government. To be a part of an integrated CASE environment is the trend of commercial software component repositories. Usually, the aim of the first one is to provide a service within a domain, organization, or area, such as ASSET for DoD, DSRS for DISA etc. This kind of repository is used in a wide scope. The aim of the second is to provide an integrated CASE environment for a software development organization. So, this kind of repository is generally a part of CASE environment and is used in a relatively narrow scope.

The long-term goal of the CAPS project [1] is to provide a distributed software component repository to support the development of prototype systems through intranet technology. So, it will combine the advantages of commercial component repositories and government supported repositories. This developing research system is an example of future software repositories.

References

[1] Luqi and Jiang Guo, "Toward Automated Retrieval for a Software Component Repository", Proceedings of IEEE International Conference and Workshop on the Engineering of Computer Based Systems (IEEE ECBS), Nashville, USA, March 7-12, 1999. Pp. 99-105.

[2] A. Mili, R. Mili, and R.Mittermeir, ``Storing and retrieving software components: A refinement based system," in Proc. 16th Int'l Conf. on Software Engineering, (Sorrento, Italy), pp. 91-100, May 1994.

[3] B. Fischer, M. Kievernagel, and W. Struckmann,``VCR: A VDM-based software component retrieval tool," in Proc. ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice, 1995.

[4] J. Penix, P. Baraona, and P. Alexander, ``Classification and retrieval of reusable components using semantic features," in Proceedings of the 10th Knowledge-Based Software Engineering Conference, pp. 131-138, Nov. 1995.

[5] A. M. Zaremski, Signature and Specification Matching. PhD thesis, Carnegie Mellon University, Jan. 1996.

[6] A. M. Zaremski and J. M. Wing, ``Specification matching of software components," in 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, Oct. 1995.

[7] J. Penix and P. Alexander, ``Design representation for automating software component reuse," in Proceedings of the first international workshop on Knowledge-Based systems for the (re)Use of Program libraries, Nov. 1995.

[8] R. McDowell, and J. Solderitsch, ``The Reusability Library Framework," Proceedings of the Unisys Defense Systems Software Engineering Symposium, January 1990.

[9] R. McDowell and K. Cassell, ``The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems," Proceedings of the 4th Annual Rome Air Development Center Knowledge-Based Software Assistant Conference, Utica, NY, September 1989.

[10] Eichmann, D., T. McGregor and D. Danley, "Integrating Structured Databases Into the Web: The MORE System," First International Conference on the World Wide Web, Geneva, Switzerland, May 25-27, 1994, pages 369-378.

[11] M. A. Durnin, K. Terry, and R. Sullins, ``Establishing a Repository for Enterprise Wide Software Reuse," in Proceedings Fifth Annual Workshop on Software Reuse Education and Training, July 29 - August 1 1996.

[12] G. Arango and R. Prieto-Diaz, "Domain Analysis Concepts and Research Directions", Domain Analysis and Software System Modeling, R. Prieto-Diaz and G. Arango eds., IEEE Computer Society, 1991.

[13] J.-J. Jeng and B. H. C. Cheng, ``A formal approach to using more general components," in Proceedings of the 9th Knowledge-Based Software Engineering Conference, pp. 90-97, September 1994.

[14] DSRS - Defense Technology for Adaptable, Reliable Systems URL:http://ssed1.ims.disa.mil/srp/dsrspage.html

[15] STARS - Software Technology for Adaptable, Reliable Systems URL: http://www.stars.ballston.paramax.com/index.html

[16] D. E. Perry and S. S. Popovitch, "Inquire: Predicate-based use and reuse," in Proceedings of the 8th Knowledge-Based Software Engineering Conference, pp. 144-151, September 1993.

[17] D. Garlan, "Research Directions in Software Architecture", ACM Computing Surveys, 27(2), June 1995.

[18] R. Girardi, "Towards Effective Software Abstractions for Application Engineering", in Procs. NASA Focus on Reuse workshop, Sept. 1996.

[19] Asset Library Open Architecture Framework, Version 1.2; STARS-TC-04041/001/02; 14 August 1992.

[20] ELSA - Electronic Library Services & Applications URL: http://rbse.mountain.net/ELSA/

[21] L. S. Levy, "A metaprogramming method and its economic justification", IEEE Trans. Softw. Eng. SE-12(2), Feb. 1996, pp. 272-277.

[22] R. Girardi and B. Ibrahim, "Using English to Retrieve Software", The Journal of Systems and Software, Special Issue on Software Reusability September 1995.

[23] R. Girardi, "Classification and Retrieval of Software through their Descriptions in Natural Language", Technical report, University of Geneva - CUI, December 1995.

[24] R. T. Price and R. Girardi. A class retrieval tool for an object-oriented environment. In Procs. 3rd Conf. Technology on object-Oriented Languages and Systems, pages 26-36, November 1990.

[25] M. Simos, "The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse," Proceedings of 1988 National Institute for Software Quality and Productivity (NISQP) Conference on Software Reusability, April 1988, pp. E-1 through E-25.

[26] ASSET - Asset Source for Software Engineering Technology URL: http://source.asset.com/asset.html

[27] PAL - Public Ada Library URL: http://web.cnam.fr/Languages/Ada/PAL/

[28] CARDS - Comprehensive Approach to Reusable Defense Software URL: http://dealer.cards.com/

[29] COSMIC - NASA's Software Technology Transfer Center URL: http://www.cosmic.uga.edu/

[30] J.-J. Jeng and B. H. C. Cheng, ``Using formal methods to construct a software library," in Proceedings of 4th European Software Engineering Conference, Lecture Notes in Computer Science, vol. 717, pp. 397-417, September 1993.

[31] +1 Software Engineeering Corporate Mission, URL http://www.plus-one.com/company .html

[32] Elisabetta Morandin, "SALMS v5.1: A System for Classifying, Describing, and Querying about Reusable Software Assets", The Proceedings of 5th International Conference on Software Reuse (ICSR '98).

[33] Project Management Tool Suite System (Automated Software Reuse Repository), URL http://wv.ewa.com-/srr_overview.html

[34] Universal Repository, URL http://www.marketplace.unisys.com/urep/

[35] Reuse Library Toolset of EVB Software Engineering, http://gopher.metronet.com:70/0/newprod/by-vendor/E-/evb_software_e/941208.01

[36] The HSTX Software Reuse Repository, selsvr.stx.com/~eryq/swreuse/home.html

[37] STARS Q9 BASELINE Ada LIBRARY, Technical Reports on the Software Reuse CFCSE-IC http://dii-sw.ncr.disa.mil/ReuseIC/guidelines/ReusabilityGuidelines.html

[38] Robert Rutherford, "Reuse on I-CASE," Proceedings of Fifth Annual Workshop on Software Reuse Education and Training, July 29, 1996 - August 1, 1996. http://www.asset.com/WSRD/conferences/proceedings/summary/-summary.html

[39] Luqi, and M. Ketabchi, "A Computer-Aided Prototyping System," IEEE Transactions on Software Engineering, October 1988.

[40] Robert Steigerwald, Luqi, and John McDowell, "CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping," Information and Software Technology, pp. 698-705, 1991.

[41] Scott Dolgoff, "Automated Interface for Retrieving Reusable Software Components", Master's Thesis, Naval Postgraduate School, September 1992.

[42] Doan Nguyen, "An Architectural Model for Software Component Search", Ph.D. Dissertation, Naval Postgraduate School, December 1995.

[43] Jeffrey Herman, "Improving Syntactic Matching for Multi-Level Filtering," M.S. Thesis, Naval Postgraduate School, September 1997.

[44] Rubin Prieto-Diaz, "Implementing Faceted Classification for Software Reuse", Communication of the ACM, pp. 89-97, May 1991.

A Risk Assessment Model for Evolutionary Software Projects[1]

Luqi, J. Nogueira
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

Current early risk assessment techniques rely on subjective human judgments and unrealistic assumptions such as fixed requirements and work breakdown structures. This is a weak approach because different people could arrive at different conclusions from the same scenario even for projects with a stable and well-defined scope, and such projects are rare. This paper introduces a formal model to assess the risk and the duration of software projects automatically, based on objective indicators that can be measured early in the process. The model has been designed to account for significant characteristics of evolutionary software processes, such as requirement complexity, requirement volatility and organizational efficiency. The formal model based on these three indicators estimates the duration and risk of evolutionary software processes. The approach supports (a) automation of risk assessment and, (b) early estimation methods for evolutionary software processes.

1. Introduction

Software applications have grown in size and complexity covering many human activities of importance to society. The report of the President s Information Advisory Committee calls software the new physical infrastructure of the information age . Unfortunately, the ability to build software has not increased proportionately to demand [Hall, 1997. pp xv], and shortfalls in this regard are a growing concern. According to the Standish group, in 1995 84% of software projects finished over time or budget, and $80 billion - $100 billion is spent annually on cancelled projects in the US. Developing software is still a high-risk activity.

There have been many approaches to improving this situation, mostly focused on increasing productivity via improvements in technology or management. Although better productivity is certainly welcome, closer examination shows that these efforts address only half of the problem. A project gets over time or over budget if actual performance does not match estimates. Current estimation techniques are far from reliable, and tend to systematically produce overly optimistic estimates. More accurate early estimates could help reduce wasted resources associated with overruns and cancelled projects in two ways: if costs are known to be too high at the outset, the scope of the project could be reduced to enable completion within time and budget, or it could be cancelled before it starts, and instead the resources could be used to successfully complete other feasible projects.

This paper therefore focuses on improved risk assessment for software projects. We address project risks related to schedule and budget, and focus mostly on completion time of the project. Current risk assessment standards are weak because they rely on subjective human expertise, assume frozen requirements, or depend on metrics difficult to measure until it is too late. This paper describes a formal risk assessment model based on metrics and sensitive to requirements volatility. Further details can be found in [Nogueira 2000]. The model is specially suited for evolutionary prototyping and incremental software development.

Section 2 defines the problem we are addressing. Section 3 analyzes relevant previous work. Section 4 presents and evaluates our project risk model. Section 5 outlines how systematic risk assessment fits into iterative prototyping. Section 6 concludes.

[1] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA, and in part by DARPA under contract #99-F759.

2. The Problem

As the range and complexity of computer applications have grown, the cost of software development has become the major expense of computer-based systems [Boehm 1981], [Karolak 1996]. Research shows that in private industry as well as in government environments, schedule and cost overruns are tragically common [Luqi 1989, Jones 1994, Boehm 1981]. Despite improvements in tools and methodologies, there is little evidence of success in improving the process of moving from the concept to the product, and little progress has been made in managing software development projects [Hall, 1997]. Research shows that 45 percent of all the causes for delayed software deliveries are related to organizational issues [vanGenuchten 1991]. A study published by the Standish Group reveals that the number of software projects that fail has dropped from 40% in 1997 to 26% in 1999. However, the percentage of projects with cost and schedule overruns rose from 33% in 1997 to 46% in 1999 [Reel 1999].

Despite the recent improvements introduced in software processes and automated tools, risk assessment for software projects remains an unstructured problem dependent on human expertise [Boehm 1988, Hall 1997]. The acquisition and development communities, both governmental and industrial, lack systematic ways of identifying, communicating and resolving technical uncertainty [SEI 1996].

This paper explores ways to transform risk assessment into a structured problem with systematic solutions. Constructing a model to assess risk based on objectively measurable parameters that can be automatically collected and analyzed is necessary. Solving the risk assessment problem with indicators measured in the early phases would constitute a great benefit to software engineering. In these early phases, changes can be made with the least impact on the budget and schedule. The requirements phase is the crucial stage to assess risk because: a) it involves a huge amount of human interaction and communication that can be misunderstood and can be a source of errors; b) errors introduced at this phase are very expensive to correct if they are discovered late; c) the existence of software generation tools can diminish the errors in the development process if the requirements are correct; and d) requirements evolve introducing changes and maintenance along the whole life cycle.

Part of the problem is misinterpreting the importance of risk management. It is usually and incorrectly viewed as an additional activity layered on the assigned work, or worse, as an outside activity that is not part of the software process [Hall 1997, Karolak 1996]. One of the goals of our research is to integrate a risk assessment model with previous research on CAPS[2] at NPS [Ham 99]. This integration is required in order to capture metrics automatically in the context of a modern evolutionary prototyping and software development process. This should provide project managers with a more complete tool that can enable improved risk assessment without interfering with the work of a project s software engineers.

A second source of problems in risk management is the lack of tools [Karolak 1996]. The main reason for this lack of tools is that risk assessment is apparently an unstructured problem. To systematize unstructured problems it is necessary to define structured processes. Structured processes involve routine and repetitive problems for which a standard solution exists. Unstructured processes require decision-making based on a three-phase method (intelligence, design, choice) [Turban et al 1998]. An unstructured problem is one in which none of the three phases is structured. Current approaches to risk management are highly sensitive to managers perceptions and preferences, which are difficult to represent by an algorithm. Depending on the decision-maker's attitude towards risk, he or she can decide early with little information, or can postpone the decision, gaining time to obtain more information, but losing some control.

A third source of risk management problems is the confusion created by the informal use of terms. Often, the software engineering community (and most parts of the project management

[2] CAPS stands for Computer Aided Prototyping System [Luqi 1988].

community [Wideman 1992]) uses the term "risk" casually. This term is often used to describe different concepts. It is erroneously used as a synonym of "uncertainty" and "threat" [SEI 1996, Hall 1997, Karolak, 1996]. Generally, software risk is viewed as a measure of the likelihood of an unsatisfactory outcome and a loss affecting the software from different points of view: project, process, and product [Hall 1997, SEI 1996]. However, this definition of risk is misleading because it confounds the concepts of risk and uncertainty. In general, most parts of decision-making in software processes are under uncertainty rather than under risk. Uncertainty is a situation in which the probability distribution for the possible outcomes is not known.

In this paper the term "risk" is reserved to indicate the probabilistic outcome of a succession of states of nature, and the term "threat" is used to identify the dangers that can occur. We define risk to be the product of the value of an outcome times its probability of occurrence. This outcome could be either positive (gain) or negative (loss). This abstraction permits one to address not only the classical risk management issue, but also to discover opportunities leading to competitive advantage.

We address the issue of risk assessment by estimating the probability distribution for the possible outcomes of a project, based on observed values of metrics that can be measured early in the process. The metrics were chosen based on a causal analysis to identify the most important threats and a statistical analysis to choose the shape of the probability distribution and relate its parameters to readily measurable metrics.

3. Related Work

There are three main groups of research related to risk:

- **Assessing Software Risk by Measuring Reliability.** This group follows a probabilistic approach and has successfully assessed the reliability of the product [Lyu 1995, Schneidewind 1975, Musa 1998]. However, this approach addresses the reliability of the product, not the risk of failing to complete the project within budget and schedule constraints. These approaches could be used to assess risks related to failures of software projects, which are outside the scope of the current paper. A concern with these approaches is that the resulting assessments arrive too late to economically correct possible faults, because the software product is mostly complete and development resources are mostly gone at the time when reliability of the product can be assessed by testing.

- **Heuristic approaches:** Other researchers assess the risk from the beginning, in parallel with the development process. However, these approaches are less rigorous, typically subjective and weakly structured. Basically these approaches use lists of practices and checklists [SEI, 1996, Hall 1997, Charette 1997, Jones 1994] or scoring techniques [Karolak 1996]. Paradoxically, SEI defines software technical risk as a measure of the probability and severity of adverse effects in developing software that does not meet its intended functions and performance requirements [SEI, 1996]. However, the term "probability" is misleading in this case because the probability distribution is unknown.

- **Macro Model Approaches:** A third group of researchers uses well known estimation models to assess how risky a project could be. The widely used methods COCOMO [Boehm 1981], and SLIM [Putnam, 1980] both assume that the requirements will remain unchanged, and require an estimation of the size of the final product as input for the models [Londeix 1987]. This size cannot be actually measured until late in the project.

The standard tools used to control all types of projects, including PERT, CPM, and Gantt, do not consider coordination and communication overhead. Such models represent sequential interdependencies through explicit representation of precedence relationships between activities. This simplified vision of a project cannot address the dynamics created by reciprocal requirements of information in concurrent activities, exception management, and the impact of actor interactions. Since the missing factors increase time requirements, the estimates resulting from these generic project estimation models are overly optimistic.

These issues are addressed by Vit Project [Levitt 1999, Thomsen et al. 1999]. Vit Project is applicable to projects in which a) all activities in the project can be predefined; b) the organization is static, and all activities are pre-assigned to actors in the static organization; c) the exceptions to activities result in extra work volume for the predefined activities and are carried out by the pre-assigned actors; and d) actors are assumed to have congruent goals. The model is well suited for simulating organizations that deal with great amounts of information processing and coordination. Such characteristics are extremely relevant in software processes [Boehm, 1981]. However, this approach requires a fixed work breakdown structure, and therefore does not apply at the early stages when requirements are changing and the set of tasks comprising the project are still uncertain.

By using informal risk assessment models, using estimation models based on optimistic assumptions that require parameters difficult to provide until late, and using optimistic project control tools, project managers condemn themselves to overrun schedules and cost.

4. The Proposed Project Risk Model

Our approach is based on metrics automatically collectable from the engineering database from near the beginning of the development. The indicators used are Requirements Volatility (RV), Complexity (CX), and Efficiency (EF).

Requirement Volatility (RV): RV is a measure of three characteristics of the requirements: a) the Birth-Rate (BR), that is the percentage of new requirements incorporated in each cycle of the evolution process; b) the Death-Rate (DR), that is the percentage of requirements dropped in each cycle; and c) the Change-Rate (CR) defined as the percentage of requirements changed from the previous version. A change in one requirement is modeled as a birth of a new requirement and the death of another, so that CR is included in the measured values of BR and DR. RV is calculated as follows: RV = BR + DR.

Complexity (CX): Complexity of the requirements is measured from a formal specification. A requirements representation that supports computer-aided prototyping, such as PSDL [Luqi 1996], is useful in the context of evolutionary prototyping. We define a complexity metric called Large Granularity Complexity (LGC) that is calculated as follows: LGC = O + D + T, where for PSDL O is the number of atomic operators (functions or state machines), D is the number of atomic data streams (data connections between operators), and T is the number of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. This is a measure of the complexity of the prototype architecture, similar in spirit to function points but more suitable for modeling embedded and real-time systems. The measure can also be applied to other modeling notations that represent modules, data connections, and abstract data types or classes. We found a strong correlation between the complexity measured in LGC and the size of PSDL specifications (correlation coefficient R = 0.996). Most important, we also found a strong correlation (R = 0.898) between the complexity measured in LGC and the size of the final product expressed in non-comment lines of Ada code, including both the code automatically created by the generator and the code manually introduced by the programmers.

Efficiency (EF): The efficiency of the organization is measured using a direct observation of the use of time. EF is calculated as a ratio between the time dedicated to direct labor and the idle time: EF = Direct Labor Time / Idle Time. We found that this easily measurable quantity was a good discriminator between high team productivity and low team productivity in a set of simulated software projects [Nogueira 2000].

We validated and calibrated our model with a series of simulated software projects using Vit Project. This tool was chosen because of the inclusion of communications and exceptions in its project dynamics model, and because it has been extensively validated for many types of engineering projects, including software engineering projects. The input parameters for the simulated scenarios were RV, EF and CX, and the observed output was the development time. Given that the proposed model uses parameters collected during the early phases and given that Vit Project requires a complete breakdown structure of the project, which can be done only in the late phases, there was a considerable time gap between the two measurements. This time gap is less than for a post-mortem analysis, but it is sufficient for model calibration and validation purposes.

The simulation results were analyzed statistically, with the finding that the Weibull probability distribution was the best fit for all the samples. A random variable x is said to have a Weibull distribution with parameters α, β and γ (with $\alpha > 0$, $\beta > 0$) if the probability distribution function (pdf) and cumulative distribution function (cdf) of x are respectively:

$$\text{pdf: } f(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ (\alpha/\beta^{\alpha})\,(x - \gamma)^{\alpha-1} \exp(-((x - \gamma)/\beta)^{\alpha}), & x \geq \gamma \end{cases}$$

$$\text{cdf: } F(x; \alpha, \beta, \gamma) = \begin{cases} 0, & x < \gamma \\ 1 - \exp(-((x - \gamma) / \beta)^{\alpha}) & x \geq \gamma. \end{cases}$$

The random variable under study, x, can be interpreted as development time in our context. The shape parameter α controls the skew of the pdf, which is not symmetric. We found that this is mostly related to the efficiency of the organization (EF). The scale parameter β stretches or compresses the graph in the *x* direction. We found that this parameter is related to the efficiency (EF), requirements volatility (RV), and complexity (CX) measured in LGC. The shifting parameter γ is shifts the origin of the curves to the right. We found that it is mostly related to the complexity measured in LGC.

Based on best fit to our simulation results, the model parameters can be derived from the project metrics using the following algorithm:

```
If (EF > 2.0)    then $\alpha$ = 1.95;
                      $\gamma$ = 22 * 0.32*(13*ln(LGC)–82);
                      $\beta$ = $\gamma$ /(5.71+(RV-20)*0.046);
                 else $\alpha$ = 2.5;
                      $\gamma$ = 22 * 0.85*(13*ln(LGC)–82);
                      $\beta$ = $\gamma$ /(5.47-(RV-20)*0.114);
                 end if;
```

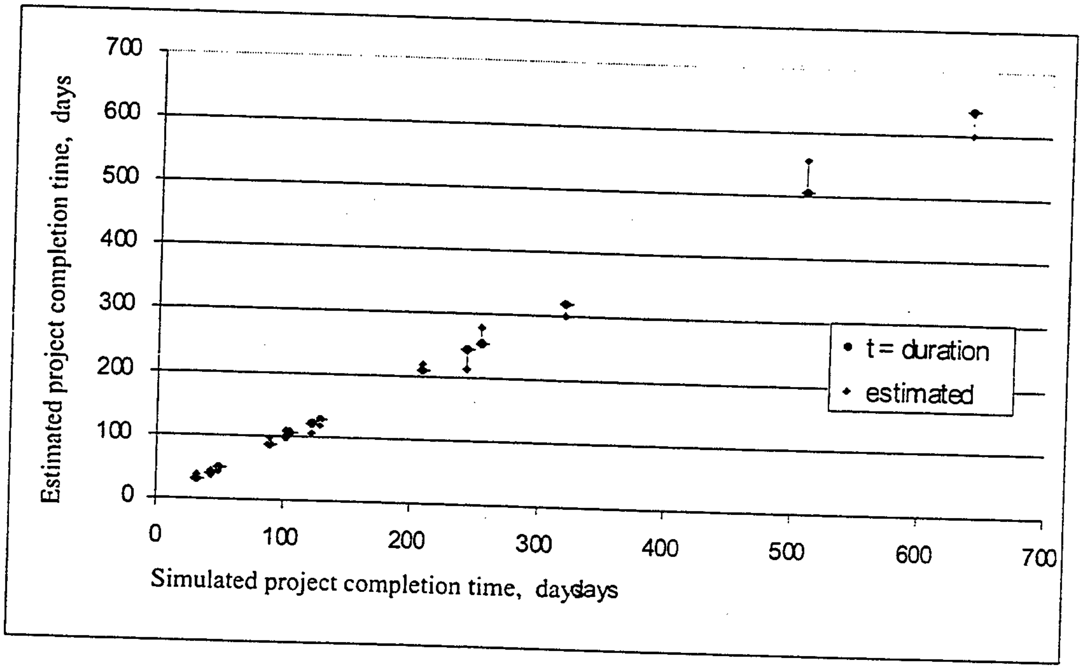
The model estimates the following cumulative probability distribution for project completion on or before time x:

$$P(x) = 1 - \exp(-(((x - \gamma)/\beta)^{\alpha})) \quad \text{// where x is time in days}$$

This equation can be inverted to obtain the schedule length needed to have a probability P of completing within schedule, with the following result.

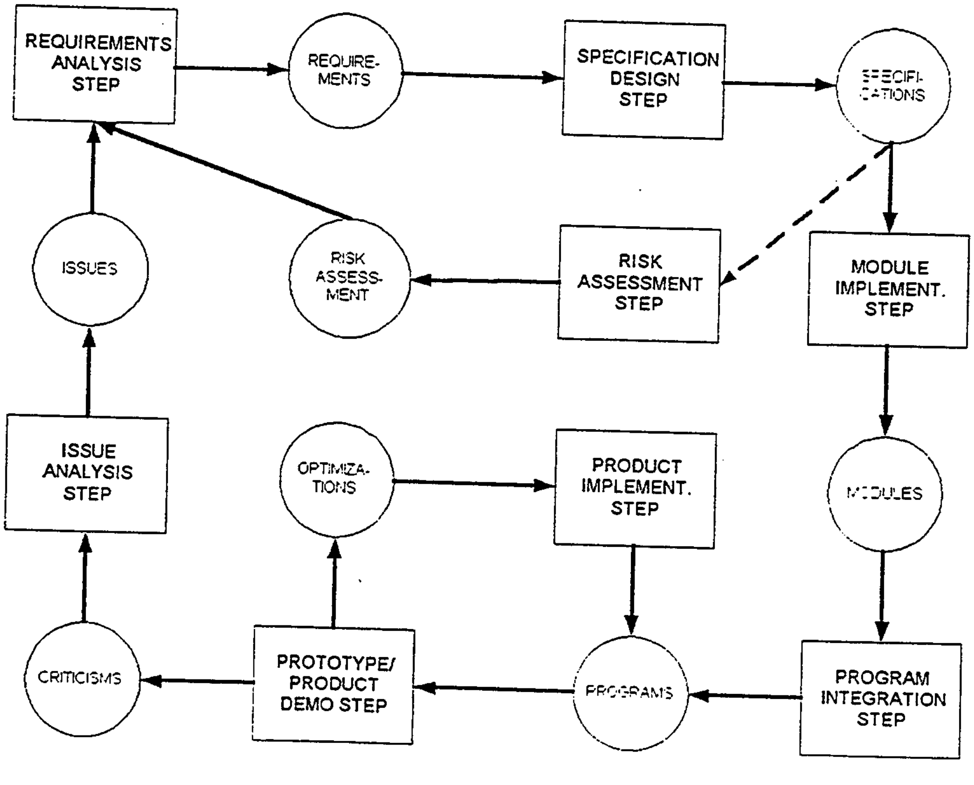
$$x = \gamma + \beta(-\ln(1-P))^{1/\alpha}$$

The probability P can be interpreted as a degree of confidence in the ability of the project to successfully complete within a schedule of length x. Applying the above equation to estimate the development time needed for a 95% chance of completion within schedule for 16 different scenarios simulated using Vit Project, we observed a standard error of 22 days. The worst case was an error of 60 days for a project of 520 days (12%). The comparison of estimated time and simulated time is shown below.



5. Integrating Risk Assessment into Prototyping

The model presented in the previous section is designed to support an iterative prototyping and software development process. In this process, an initial problem statement. a prototype demo or problem reports from a deployed software product trigger an issue analysis, followed by formulation of proposed requirements changes, and specification of a proposed adjustment to the software requirements, which can be initially empty. At this point in each cycle, the project manager should perform a risk assessment step. The results of the risk assessment step guide the degree of detail to which requirements enhancements are demonstrated, and the set of requirements issues to be considered in the next prototyping cycle, if any.



The first measurement-based risk assessment step can be performed after specification of the first version of the prototype architecture, based on the requirements volatility, LGC and efficiency measurements from the steps just performed.

In cases where risk assessments are required even earlier, before any prototyping has been done, estimates of team efficiency and requirements volatility can be based on measurements of similar past projects, and initial complexity estimates can be based on subjective guesswork of the kind currently used in the macro model approaches. This kind of estimate may be less reliable than those based solely on measurements, but it can provide a principled and reasonably accurate basis for deciding whether or not to start a prototyping process to determine the requirements for a proposed development project. Thus parts of our approach can be used truly at the very beginning of the process.

If a prototyping effort is approved, early measurements of the process could be used to refine the initial estimates of the model parameters using Bayesian methods, thus providing a balanced and systematic transition from subjective guesswork, coded as an *a priori* distribution, to assessments increasingly based on systematic measurement. Such an approach also supports incorporation and systematic refinement of measurements from previous cycles of the iterative prototyping process.

The results of risk assessment can provide guidance on the degree to which the project can afford to explore requirements enhancements requested by the customers. It can also help customers or marketing departments to decide how much they really want possible improvements, in the context of the resulting time and cost estimates. Systematic cost/benefit analysis becomes possible only with the availability of reasonably accurate estimates.

The risk assessment step can thus provide a balancing force to stabilize the requirements formulation process. In the absence of information on how much potential enhancements will cost, stakeholders are prone to unrealistic requirements amplification — of course they would always like to have a better system, no matter how good the existing one is, if you do not ask them to pay for the improvements. The proposed risk assessment steps can provide a realistic basis for incorporating time and cost constraints and cost/benefit tradeoffs early in the process, when the situation is fluid and many options are open.

This process refinement provides some additional insight into the dynamics of iterative prototyping: the iterative process should stop when the customers have determined what requirements they can afford to realize, and which of many possible improvements they will be willing to pay for, if any. It is not necessarily the case that the set of criticisms elicited by the final round of prototype demonstrations is empty — that is true only in an idealized world with adequate budgets and patient customers.

6. Conclusion

This paper introduces a formal risk assessment model for software projects based on probabilities and metrics automatically collectable from the project baseline. The approach enables a project manager to evaluate the probability of success of the project very early in the life cycle, during an iterative requirements formulation process, based on well-defined measurements rather than just guesswork or subjective judgments.

For more than twenty years, estimation standards have been characterized by a common limitation: the requirements should be frozen in order to make estimates. This model presented in this paper removes this important limitation, facing the reality that requirements are inherently variable.

The model is perfectly suited for any evolutionary software process because it follows the same philosophy. The risk assessment and estimation steps are conducted at each evolutionary cycle with increasing knowledge and decreasing variance. The research formalizes an improvement in the evolutionary software process, introducing a risk assessment step that can be automated, and that can help shape the planning of the project in the early stages when there is still substantial freedom to allocate available time and budget.

References

[Boehm 1981] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Boehm 1988] B. Boehm, A Spiral Model of Software Development and Enhancement, *Computer*, May 1988.

[Charette 1997] R. Charette, K. Adams, & M. White, Managing Risk in Software Maintenance, *IEEE Software*, May-June, 1997.

[Gilb 1977] T. Gilb, *Software Metrics*, Winthrop Publishers, Inc., 1977.

[Hall 1997] E. Hall, Managing Risk, *Methods for Software Systems Development*, Addison Wesley, 1997.

[Harn 1999] M. Harn, V. Berzins, Luqi, Computer-Aided Software Evolution Based on a Formal Model, *Proceedings of the Thirteenth International Conference on Systems Engineering*, Las Vegas, Nevada, August 9-12, 1999, pp. CS: 55-60.

[Jones 1994] C. Jones, *Assessment and Control of Software Risks*, Yourdon Press Prentice Hall, 1994.

[Karolak 1996] D. Karolak, Software *Engineering Management*, IEEE Computer Society Press, 1996.

[Levitt 1999] R. Levitt, *The ViteProject Handbook: A User's Guide to Modeling and Analyzing Project Work Processes and Organizations*, Vit ' 1999.

[Londeix 1987] B. Londeix, *Cost Estimation for Software Development*, Addison-Wesley, 1987.

[Luqi 1988] Luqi, M. Ketabchi, A Computer Aided Prototyping System, *IEEE Software*, Vol. 5, No. 2, p. 66-72, March 1988.

[Luqi 1989] Luqi, Software Evolution Through Rapid Prototyping, *IEEE Computer*, May 1989.

[Luqi 1996] Luqi, Special Issue: Computer-Aided Prototyping, *Journal of Systems Integration*, Vol. 6, Nos. 1-2, March 1996.

[Lyu 1995] M. Lyu, *Software Reliability Engineering*, IEEE Computer Society Press. 1995.

[Musa 1998] J. Musa, Software *Reliability Engineering: More Reliable Software, Faster Development and Testing*, McGraw-Hill, 1998.

[Nogueira 2000] J. Nogueira, *A Formal Risk Assessment Model for Software Projects*, Ph.D. Dissertation, Naval Postgraduate School, 2000.

[Putnam 1980] L. Putnam, *Software Cost Estimating and Life-cycle Control: Getting the Software Numbers*, IEEE Computer Society Press, 1980.

[Reel 1999] J. Reel, *Critical Success Factors in Software Projects*, IEEE Software, May - June 1999.

[SEI 1996] Software Engineering Institute, Software Risk Management, Technical Report CMU/SEI-96-TR-012, June 1996.

[Schneidewind 1975] N. Schneidewind, Analysis of Error Processes in Computer Software, *Proceedings of the International Conference on Reliable Software*, IEEE Computer Society, 21-23 April 1975, p 337-346.

[Turban et al 1998] E. Turban and J. Aronson, *Decision Support Systems and Intelligent Systems*, Prentice Hall, 1998.

[vanGenuchten 1991] M. van Genuchten, Why is Software Late? An Empirical Study of the Reasons for Delay in Software Development, *IEEE Transactions on Software Engineering*, June, 1991.

[Wideman 1992] R. Wideman, *Risk Management: A Guide to Managing Project Risk Opportunities*, Project Management Institute, 1992.

[Alt+99] Althoff, K., Birk, A., Hartkopf, S., Müller, W., Nick, M., Surmann, D., and Tautz, C., "Managing Software Engineering Experience for Comprehensive Reuse", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

[Ber+97] Berzins, V., Ibrahim, O., Luqi: "A Requirements Evolution Model for Computer Aided Prototyping", Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, Madrid, Spain, 1997.

[BirTau98] Birk, A. and Tautz, C., "Knowledge Management of Software Engineering Lessons Learned", Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering, San Francisco Bay, California, USA, 1998.

[BroRun99] Broomé, M. and Runeson, P., "Technical Requirements for the Implementation of an Experience Base", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

[FIPA] Foundation for Intelligent Physical Agents. http://www.cselt.stet.it/fipa/.

[Har+99] Harn, M., Berzins, V., and Luqi, "Computer-Aided Software Evolution Based on a Formal Model", Proceedings of the 13th International. Conference on Systems Engineering, Las Vegas, NV, USA, 1999.

[IEEE1220] IEEE Std 1220-1998, *IEEE Standard for Application and Management of the Systems Engineering Process*, Institute of Electrical and Electronics Engineers, 1998.

[Rob+00] Robinson, M., Kovalainen, M., and Auramäki, E., "Diary as Dialogue in Papermill Process Control", *Communications of the ACM*, Vol. 43, No. 1, January 2000.

[SEI99] Software Engineering Institute, Capability Maturity Model®-Integrated-Systems/Software Engineering: Staged Representation - Volume 1, Version 0.2b, 1999.

[Sta99] Statz, J., "Leverage Your Lessons", *IEEE Software*, Vol. 16. No. 2, IEEE, 1999.

[WalUng91] Walsh, J. and Ungson, G., "Organizational Memory", *Academy of Management Review*", Vol. 16., No. 1, January 1991.

[Wan+99] Wangenheim, C., Althoff, K., and Barcia, R., "Intelligent Retrieval of Software Engineering Experienceware", Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering, Kaiserslautern, Germany, 1999.

Evolutionary Computer Aided Prototyping System (CAPS)*

Luqi, V. Berzins, M. Shing, R. Riehle and J. Nogueira
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5118
{luqi, berzins, mantak}@cs.nps.navy.mil
{rdriehle, jcnogue}@nps.navy.mil

Abstract

This paper describes a distributed development environment, CAPS (Computer-Aided Prototyping System), to support rapid prototyping and automatic generation of source code based on designer specifications in an evolutionary software development process. The CAPS system uses a fifth-generation prototyping language to model the communication structure, timing constraints, I/O control, and data buffering that comprise the requirements for an embedded software system. The language supports the specification of hard real-time systems with reusable components from domain specific component libraries. CAPS has been used successfully as a research tool in prototyping large real-time control systems (e.g. the command-and-control station, cruise missile flight control system, missile defense systems) and demonstrated its capability to support the development of large complex embedded software.

1. Introduction

Studies have shown that early parts of the system development cycle such as requirements and design specifications are especially prone to errors [1]. Problems originated in the early stages often have a lasting influence on the reliability, safety and cost of the system. Evolutionary prototyping offers an iterative approach to requirements engineering to alleviate the problems of uncertainty, ambiguity and inconsistency inherent in the process. Moreover, prototyping can improve the capture of change in requirements and assumptions during the development process. This effect is particularly observed in projects involving multiple stakeholders with different points of view [4, 15].

Evolutionary driven computer aided software engineering (CASE) tools for computer-aided prototyping provide logical assessment of the consistency and clarity of requirements and specifications. Prototypes facilitate the requirements phase in any type of software projects. Particularly, in real-time applications where severe time constraints impose more challenges, the use of prototypes helps to describe the requirements in a clear, precise, consistent and executable format. Prototypes can demonstrate system scenarios to the affected parties as a way to: a) collect criticisms and feedback for updated requirements; b) early detection of deviations from users' expectations; c) trace the evolution of the requirements; d) improve the communication and integration of the users and the development personnel; and e) provide

* This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA.

early warning of mismatches between proposed software architectures and the conceptual structure of requirements.

The benefits of prototyping are widely accepted. All modern life cycle models such as Boehm's spiral [2]. Luqi's graph model [9], rapid application development (RAD), etc. are based on prototyping. Experience suggests that building and integrating software by mechanically processable formal models leads to cheaper, earlier and more reliable products [13]. Bernstein estimated that for every dollar invested in prototyping, one can expect a $1.40 return within the life cycle of the system development [3]. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Software for rapid and inexpensive construction and modification of prototypes makes it feasible [10, 11].

2. The Computer Aided Prototyping System (CAPS)

The Computer-Aided System (CAPS), a research tool developed at the Naval Postgraduate School, is an integrated set of software tools that generate source programs directly from high level requirements specifications (Figure 1) [8]. CAPS provides the following kinds of support to the prototype designer: a) timing feasibility checking via the scheduler; b) consistency checking and automated assistance for project planning, configuration management, scheduling, designer task assignment, and project completion date estimation via the Evolution Control system; c) computer-aided design completion via the editors; d) computer-aided software reuse via the software base; and e) automatic generation of wrapper and glue code via the execution support system.

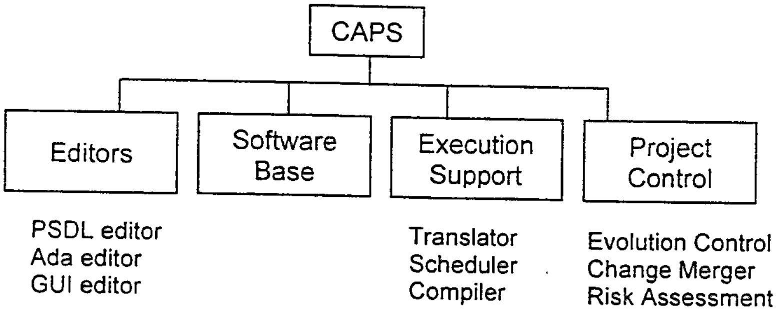


Figure 1. The CAPS rapid prototyping environment

The efficacy of CAPS has been demonstrated in many research projects (e.g. the command-and-control station, cruise missile flight control system, SIDS wireless acoustic monitor, and missile defense systems) at the Naval Postgraduate School and other facilities.

There are four major stages in the CAPS rapid prototyping process: software system design, construction, execution, and requirements evaluation/modification (Figure 2).

The initial prototype design starts with an analysis of the problem and a decision about which parts of the proposed system are to be prototyped. Requirements for the prototype are then generated, either informally (e.g. English) or in some formal notation. These requirements may be refined by asking users to verify their completeness and correctness.

After some requirements analysis, the designer uses the CAPS PSDL editor to draw dataflow diagrams annotated with nonprocedural control constraints as part of the specification of a hierarchically structured prototype, resulting in a preliminary, top-level design free from programming level details. The user may continue to decompose any software module until its components can be realized via reusable components drawn from the software base or new atomic components.

This prototype is then translated into the target programming language for execution and evaluation. Debugging and modification utilize a design database that assists the designers in managing the design history and coordinating change, as well as other tools shown in Figure 1.

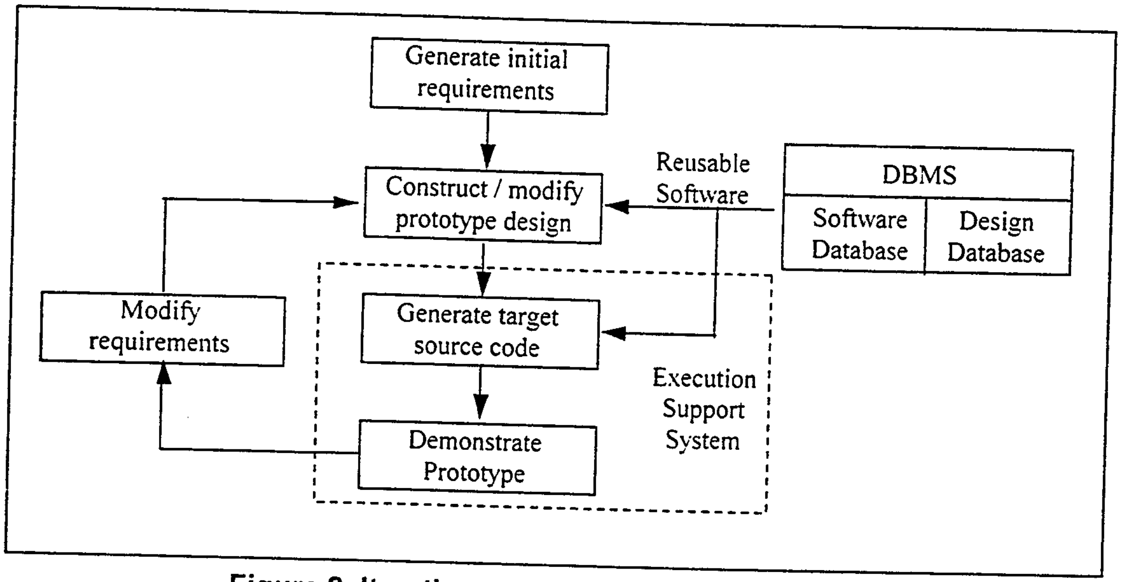


Figure 2. Iterative prototyping process in CAPS

3. Application of CAPS in an evolutionary software process

3.1 CAPS as a Requirements Engineering tool

The requirements for a software system are expressed at different levels of abstraction and with different degrees of formality. The highest level requirements are usually informal and imprecise, but they are understood best by the customers. The lower levels are more technical, precise, and better suited for the needs of the system analysts and designers, but they are further removed from the user's experiences and less well understood by the customers. Because of the differences in the kinds of descriptions needed by the customers and developers, it is not likely that any single representation for requirements can be the "best" one for supporting the entire software development process. CAPS provides the necessary means to bridge the communication gap between the customers and developers. The CAPS tools are based on the Prototype System Description Language (PSDL), which is designed specifically for specifying hard real-time systems [6, 7]. It has a rich set of timing specification features and offers a common baseline from which users and software engineers describe requirements. The PSDL descriptions of the prototype produced by the PSDL editor are very formal, precise and unambiguous, meeting the needs of the system analysts and designers. The demonstrated behavior of the executable prototype, on the other hand, provides concrete information for the customer to assess the validity of the high level requirements and to refine them if necessary.

3.2 CAPS as a System Testing and Integration tool

Unlike throw-away prototypes, the process supported by CAPS provides requirements and designs in a form that can be used in construction of the operational system. The prototype provides an executable representation of system requirements that can be used for comparison during system testing. The existence of a flexible prototype can significantly ease system testing and integration. When final implementations of subsystems are delivered, integration and testing can begin before all of the subsystems are complete by combining the final versions of the completed subsystems with prototype versions of the parts that are still being developed.

3.3 CAPS as an Acquisition tool

Decisions about awarding contracts for building hard real-time systems are risky because there is little objective basis for determining whether a proposed contract will benefit the sponsor at the time when those decisions must be made. It is also very difficult to determine whether a delivered system meets its requirements. CAPS, besides being a useful tool to the hard real-time system developers, is also very useful to the customers. Acquisition managers can use CAPS to ensure that acquisition efforts stay on track and that contractors deliver what they promise. CAPS enables validation of requirements via prototyping demonstration, greatly reducing the risk of contracting for real-time systems.

3.4 CAPS as a Risk Assessment tool

The use of prototypes introduces a problem for project planning because of the uncertain number of prototyping cycles required before constructing the product and the amount of complexity that should be covered at each cycle. Many existing project management and estimation techniques are based on linear layouts of activities. CPM and PERT techniques are not well suited to deal with cycles because they are based on acyclic digraphs.

Figure 3 shows a typical evolutionary prototyping software process that is a directed graph with two cycles. Initially, the analysts collect a set of issues, which represent concerns and preliminary goals of the customers, and transform them into a more elaborated level of description called requirements using a requirements analysis step. The requirements are transformed into specifications, probably in PSDL, during the specification design step. In the module implementation step the specifications are automatically converted into code using an appropriate CASE tool such CAPS. The program integration step transforms the modules obtained by the generator are integrated into a program, possibly adding code created by programmers and reusable components. This step includes integration testing and debugging. The program is demonstrated to the customer as a prototype. There are two possible outcomes: a) the customer is not satisfied and introduces criticisms, or b) the product matches the needs and expectations of the customer. In the first case, the process continues by analyzing the criticisms during an issue analysis step that produces new issues closing the external cycle in the graph. In the second case, the prototype contains all the required functionality, so a set of optimizations is introduced during a product implementation step. The resulting product is presented again to the customer during a product demo step closing the internal cycle of the graph.

We improved the evolutionary prototyping software process by introducing a new vertex in the graph to contain the risk assessment step (Figure 4) [14]. A risk assessment step can be automatically done after the completion of the specifications. CAPS provides the automation needed to derive the complexity of the product from the PSDL specifications. This derivation will be used together with personnel and organizational information, and with metrics of requirements collected from the baselines, to produce the risk assessment. The requirements analysis step integrates these measures with issues in the issue analysis steps.

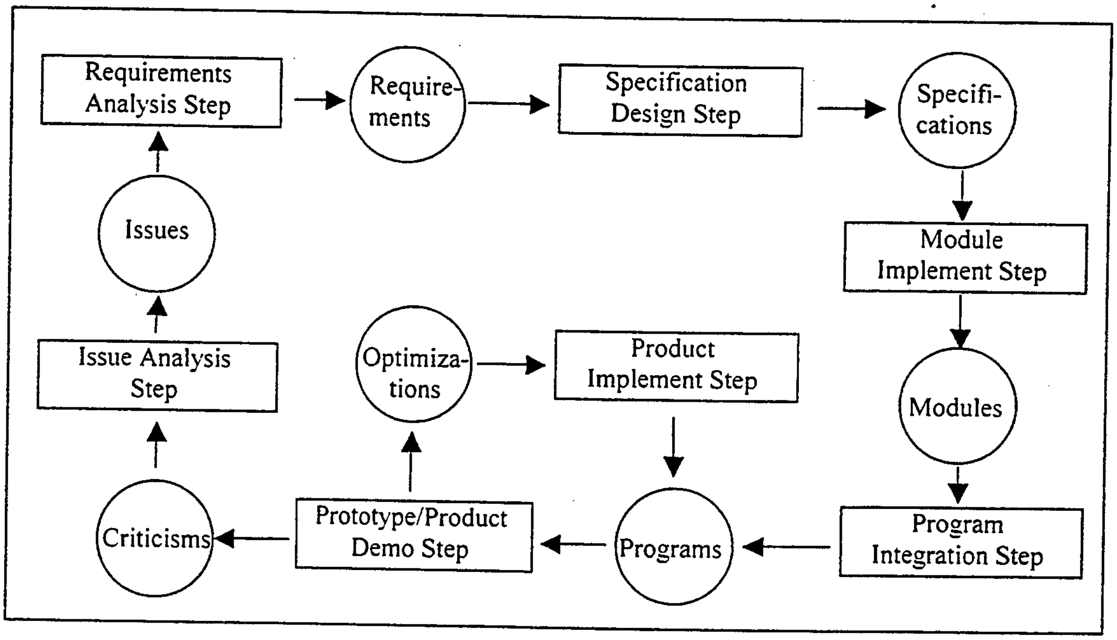


Figure 3. A typical evolutionary prototyping software process.

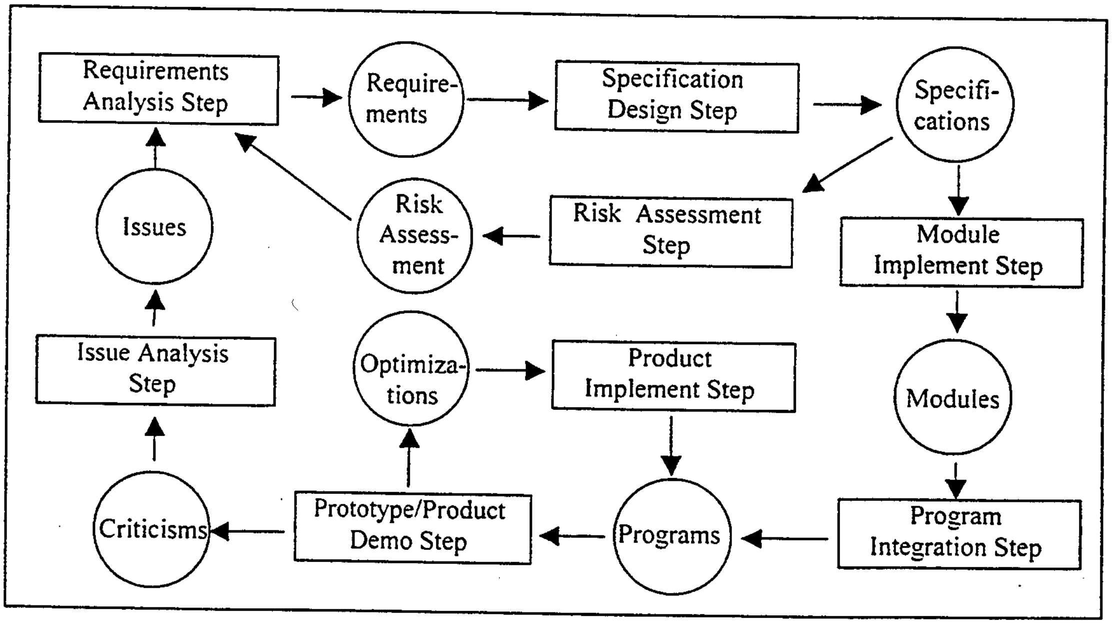


Figure 4. The improved evolutionary prototyping software process.

4. A simple example: prototyping a C3I workstation

To create a first version of a new prototype, users can select "New" from the "Prototype" pull-down menu of the CAPS main interface. The user will then be asked to provide the name of the new prototype (say "c3i_system") and the CAPS PSDL editor will be automatically invoked with a single initial root operator (with a name same as that of the prototype).

CAPS allows the user to specify the prototype requirements as augmented dataflow graphs. Using the drawing tools provided by the PSDL editor, the user can create the top-level dataflow diagram of the c3i_system prototype as shown in Figure 5. The c3i_system prototype is modeled by nine modules, communicating with each other via data streams. To model the dynamic behavior of these modules, the dataflow diagram is augmented with control and timing constraints. For example, the user may want to specify that the weapons_interface module has a maximum response time of 3 seconds to handle the event triggered by the arrival of new data in the weapon_status_data stream, and only writes output to the weapon_emrep stream if the status of the weapon_status_data is damage, service_required, or out_of_ammunition. CAPS allows the user to specify these timing and control constraints using the pop-up operator property menu (Figure 6), resulting in a top-level PSDL program shown in Figure 7.

To complete the specification of the c3i_system prototype, the user must specify how each module will be implemented by choosing the implementation language for the module via the operator property menu. The implementation of a module can be in either the target programming language or PSDL. A module with an implementation in the target programming language is called an atomic operator. A module that is decomposed into a PSDL implementation is called a composite operator. Module decomposition can be done by selecting the corresponding operator in the tree-panel on the left side of the PSDL editor.

The user may choose to implement all nine modules as atomic operators (using dummy components) in the first version, so as to check out the global effects of the timing and control constraints. Then, he/she may choose to decompose the comms_interface module into more detailed subsystems and implement the sub-modules with reusable components, while leaving the others as atomic operators in the second version of the prototype, and so on.

To facilitate the testing of the prototypes, CAPS provides the user with an execution support system that consists of a translator, a scheduler and a compiler. Once the user finishes specifying the prototype, he/she can invoke the translator and the scheduler from the CAPS main interface to analyze the timing constraints for feasibility and to generate a supervisor module for each subsystem of the prototype in the target programming language. Each supervisor module consists of a set of driver procedures that realize all the control constraints, a high priority task (the static schedule) that executes the time-critical operators in a timely fashion, and a low priority dynamic schedule task that executes the non-time-critical operators when there is time available. The supervisor module also contains information that enables the compiler to incorporate all the software components required to implement the atomic operators and generate the binary code automatically. The translator/scheduler also generates the glue code needed for timely delivery of information between subsystems across the target network.

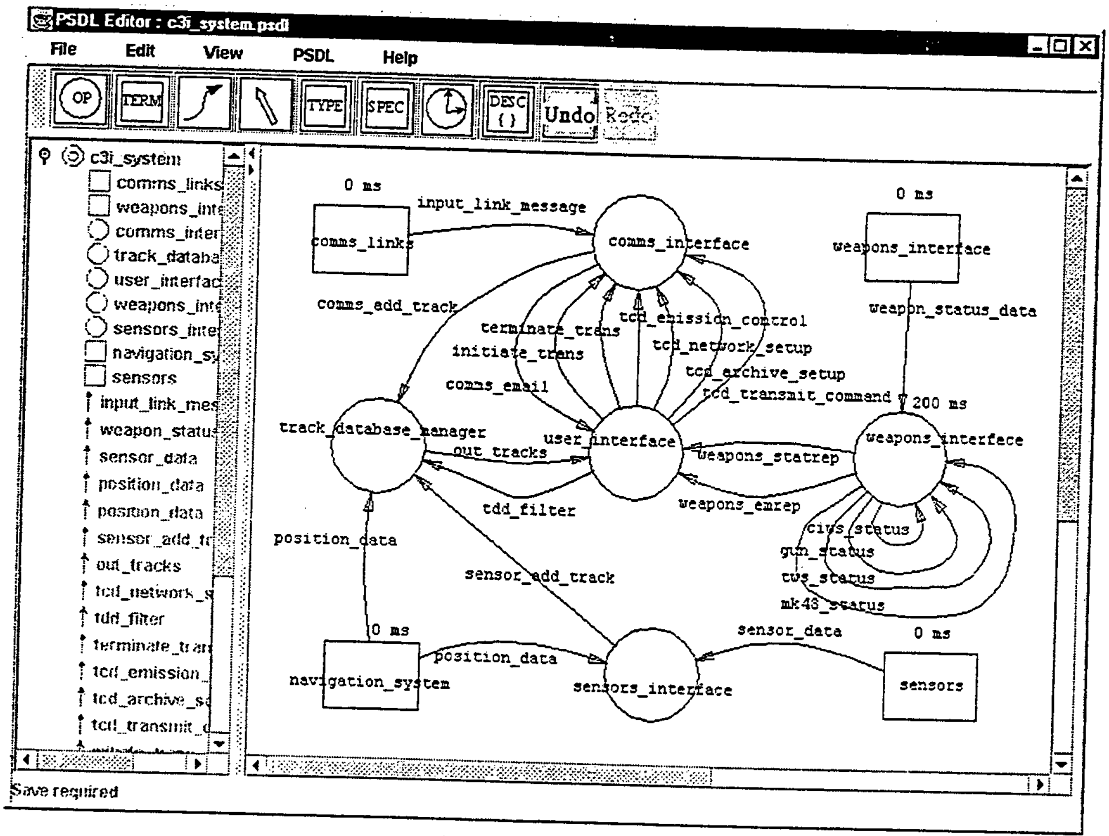
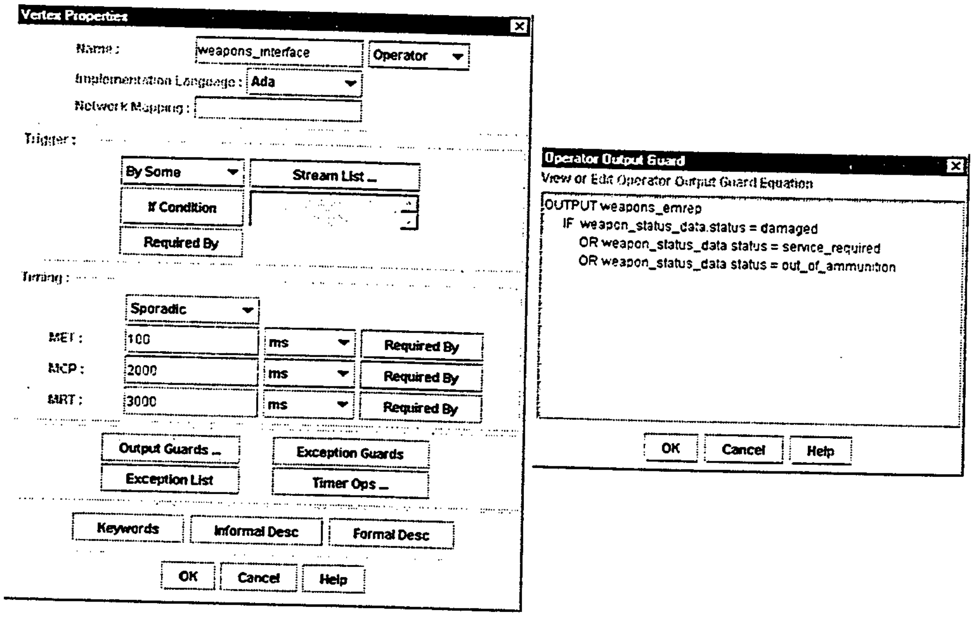


Figure 5. Top-level dataflow diagram of the c3i_system.



Vertex Properties

Name : weapons_interface Operator

Implementation Language : Ada

Network Mapping :

Trigger :

By Some | Stream List ...

If Condition

Required By

Timing :

Sporadic

MET : 100 ms Required By

MCP : 2000 ms Required By

MRT : 3000 ms Required By

Output Guards ... | Exception Guards

Exception List | Timer Ops ...

Keywords | Informal Desc | Formal Desc

OK | Cancel | Help

Operator Output Guard

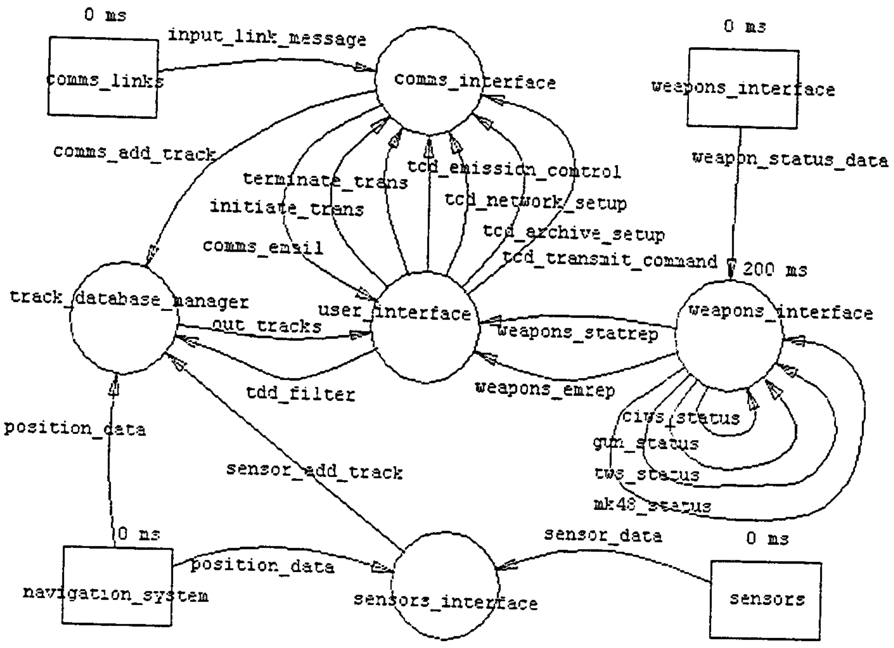
View or Edit Operator Output Guard Equation

```
OUTPUT weapons_emrep
   IF weapon_status_data.status = damaged
      OR weapon_status_data status = service_required
      OR weapon_status_data status = out_of_ammunition
```

OK | Cancel | Help

Figure 6. Pop-up operator property menu

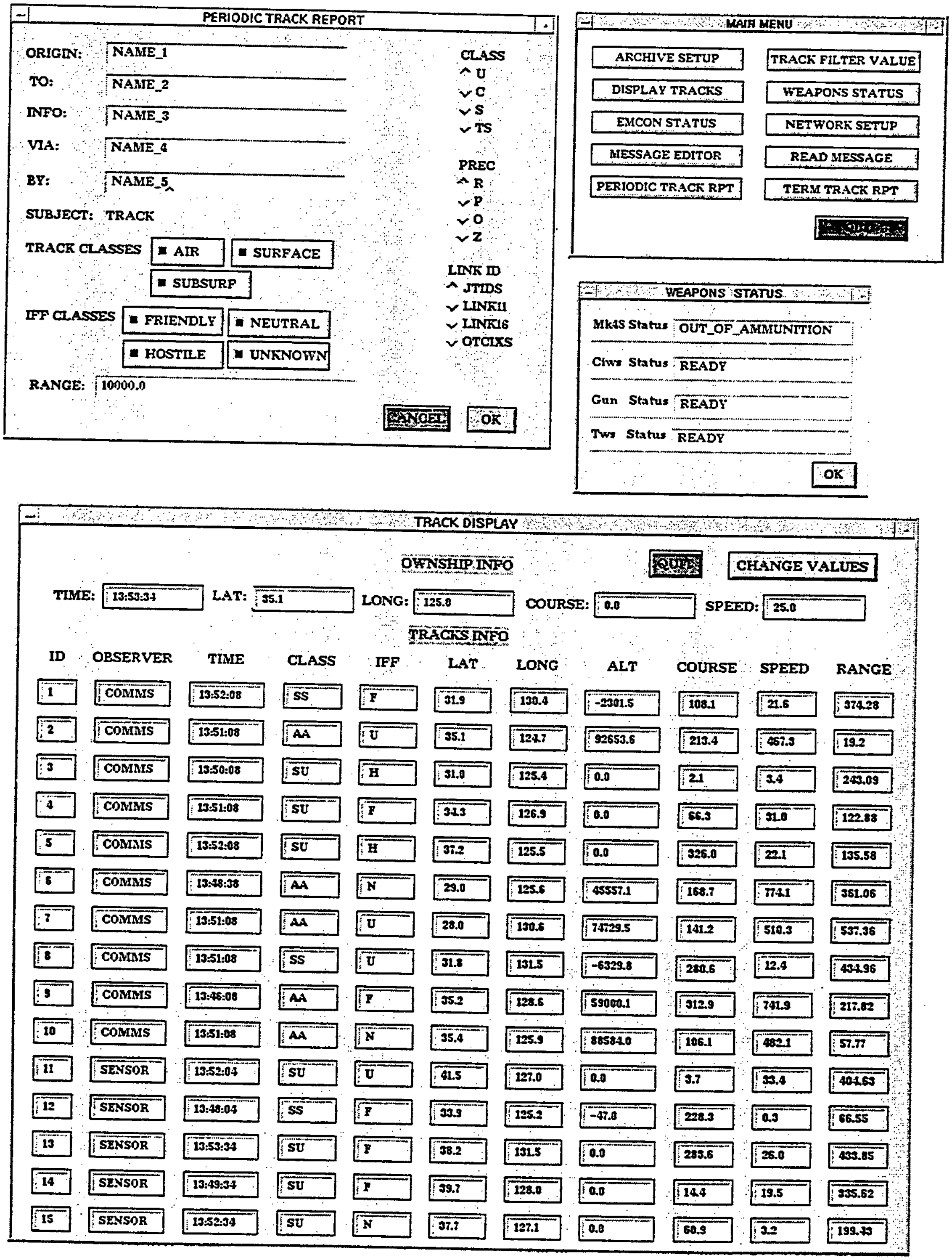
```
OPERATOR c3i_system
SPECIFICATION
     DESCRIPTION
       {This module implements a simplified version of
        a generic C3I workstation.}
END
IMPLEMENTATION
     GRAPH
```



```
DATA STREAM
-- Type declarations for the data streams in the graph go here.

CONTROL CONTRAINTS
  OPERATOR comms_links
    PERIOD 30000 MS

  OPERATOR navigation_system
    PERIOD 30000 MS

  OPERATOR sensors
    PERIOD 30000 MS

  OPERATOR weapons_systems
    PERIOD 30000 M

  OPERATOR weapons_interface
    TRIGGERED BY SOME
      weapon_status_data
    MINIMUM CALLING PERIOD 2000 MS
    MAXIMUM RESPONSE TIME 3000 MS
    OUTPUT
      weapons_emrep
      IF weapon_status_data.status
          = damaged
      OR weapon_status_data.status
          = service_required
      OR weapon_status_data.status
          = out_of_ammunition
END
```

Figure 7. Top-level specification of the c3i_system

For prototypes which require sophisticated graphic user interfaces, the CAPS main interface provides an interface editor to interactively sculpt that interface. In the final version of the c3i_system prototype, we choose to decompose the comms_interface, the track_database_manager and the user_interface modules into subsystems, resulting in hierarchical design consisting of 8 composite operators and twenty-six atomic operators. The user interface of the prototype has a total of 14 panels, four of which are shown in Figure 8. The corresponding Ada program has a total of 10.5K lines of source code. Among the 10.5K lines of code, 3.5K lines come from supervisor module that was generated automatically by the translator/scheduler and 1.7K lines that were automatically generated by the interface editor [12].



ORIGIN: NAME_1
TO: NAME_2
INFO: NAME_3
VIA: NAME_4
BY: NAME_5
SUBJECT: TRACK
TRACK CLASSES ■ AIR ■ SURFACE ■ SUBSURF
IFF CLASSES ■ FRIENDLY ■ NEUTRAL ■ HOSTILE ■ UNKNOWN
RANGE: 10000.0

CLASS
^ U
✓ C
✓ S
✓ TS

PREC
^ R
✓ P
✓ O
✓ Z

LINK ID
^ JTIDS
✓ LINK11
✓ LINK16
✓ OTCIXS

Mk45 Status OUT_OF_AMMUNITION
Ciws Status READY
Gun Status READY
Tws Status READY

OWNSHIP INFO

TIME: 13:53:34 LAT: 35.1 LONG: 125.0 COURSE: 0.0 SPEED: 25.0

TRACKS INFO

ID	OBSERVER	TIME	CLASS	IFF	LAT	LONG	ALT	COURSE	SPEED	RANGE
1	COMMS	13:52:08	SS	F	31.9	130.4	-2301.5	108.1	21.6	374.28
2	COMMS	13:51:08	AA	U	35.1	124.7	92653.6	213.4	467.3	19.2
3	COMMS	13:50:08	SU	H	31.0	125.4	0.0	2.1	3.4	243.09
4	COMMS	13:51:08	SU	F	34.3	126.9	0.0	66.3	31.0	122.88
5	COMMS	13:52:08	SU	H	37.2	125.5	0.0	326.0	22.1	135.58
6	COMMS	13:48:38	AA	N	29.0	125.6	45557.1	168.7	774.1	361.06
7	COMMS	13:51:08	AA	U	28.0	130.6	74729.5	141.2	510.3	537.36
8	COMMS	13:51:08	SS	U	31.8	131.5	-6329.8	280.6	12.4	434.96
9	COMMS	13:46:08	AA	F	35.2	128.6	59000.1	312.9	741.9	217.82
10	COMMS	13:51:08	AA	N	35.4	125.9	88584.0	106.1	482.1	57.77
11	SENSOR	13:52:04	SU	U	41.5	127.0	0.0	3.7	33.4	404.63
12	SENSOR	13:48:04	SS	F	33.9	125.2	-47.0	228.3	0.3	66.55
13	SENSOR	13:53:34	SU	F	38.2	131.5	0.0	283.6	26.0	433.85
14	SENSOR	13:49:34	SU	F	39.7	128.8	0.0	14.4	19.5	335.62
15	SENSOR	13:52:34	SU	N	37.7	127.1	0.0	60.9	3.2	199.39

Figure 8. User interface of the c3i_system

To evaluate the benefits derived from the practice of computer-aided prototyping within the software acquisition process, we conducted a case study in which we compared the cost (in dollar amounts) required to perform requirements analysis and feasibility study for the c3i system using the Mil-Std 2167A process, in which the software is coded manually, and the rapid prototyping process, where part of the code is automatically generated via CAPS [5].

We found that, even under very conservative assumptions, using the CAPS method resulted in a cost reduction of $56,300, a 27% cost saving. Taking the results of this comparison, then projecting to a mission control software system, the command and control segment (CCS), we estimated that there would be a cost saving of 12 million dollars. Applying this concept to an engineering change to a typical component of the CCS software showed a further cost savings of $25,000.

5. Conclusion

CAPS has been used successfully as a research tool in prototyping large war-fighter control systems and demonstrated its capability to support the development of large complex embedded software. Specific payoffs include:

(1) Formulate/validate requirements via prototype demonstration and user feedback,
(2) Assess feasibility of real-time system designs,
(3) Enable early testing and integration of completed subsystems,
(4) Support evolutionary system development, integration and testing,
(5) Reduce maintenance costs through systematic code generation,
(6) Produce high quality, reliable and flexible software,
(7) Avoid schedule overruns.

6. References

[1] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[2] B. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, 21(5), pp. 61-72, 1988.

[3] L. Bernstein, "Forward: Importance of Software Prototyping ", *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, 6(1). pp. 9-14. 1996.

[4] J. Conklin and M. Begeman. "GIBIS: A Hypertext Tool for Exploratory Policy Discussion", *ACM Transactions on Office Information Systems*. 6(4). pp. 303-331, 1988.

[5] M. Ellis. *Computer-Aided Prototyping Systems (CAPS) within the software acquisition process: a case study*, Master's thesis, Naval Postgraduate School, Monterey, California, June 1993.

[6] B. Kraemer, Luqi and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language", *IEEE Transaction on Software Engineering*. 19(5). pp. 453-477, 1993.

[7] Luqi, V. Berzins and R. Yeh. "A Prototyping Language for Real-Time Software", *IEEE Transaction on Software Engineering*, 14(10), pp. 1409-1423, 1988.

[8] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Software*, 5(2). pp. 66-72, 1988.

[9] Luqi, "A Graph Model for Software Evolution", *IEEE Transactions on Software Engineering*, 16(8), pp. 917-927, 1990.

[10] Luqi and W. Royce, "Status Report: Computer-Aided Prototyping", *IEEE Software*, 9(6), pp. 77-81, 1991.

[11] Luqi, "Computer-Aided Software Prototyping", *IEEE Computer*, 24(9). pp. 111-112. 1991.

[12] Luqi. "Computer-Aided Prototyping for a Command-and-Control System Using CAPS", *IEEE Software*, 9(1), pp. 56-67, 1992.

[13] Luqi and J. Goguen, "Formal methods: promises and problems", *IEEE Software*. 14(1). pp. 73-85, 1997.

[14] J. Nogueira, *A Formal Model for Risk Assessment in Software Projects*. Doctoral Dissertation, Software Engineering, Naval Postgraduate School, Sept. 2000.

[15] B. Ramesh and Luqi, "Process Knowledge Based Rapid Prototyping Requirements Engineering", *Journal of Systems Integration*, 5(2). pp. 157-177, 1995.

The Use of Computer Aided Prototyping for Re-engineering Legacy Software

Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams
J. Guo and B. Shultes

Abstract

Re-engineering is typically needed when a system performing a valuable service must change, and its current implementation can no longer support cost-effective changes. The process of re-engineering old procedural software to a modern object-oriented architecture introduces certain complexities into the software analysis process. The direct products of reverse engineering, such as requirements or design specifications, are likely to have a functionally based structure. As a result, some transformation of the recovered requirements and design specifications is necessary in order to obtain specifications for the new structures. It is often very difficult to quickly determine if the transformed specification is a true representation of the desired requirements. This paper discusses the effective use of computer-aided prototyping techniques for re-engineering legacy software, and presents results of a case study which showed that prototyping can be a valuable aid in re-engineering of legacy systems, particularly in cases where radical changes to system conceptualization and software structure are needed. The CAPS system enabled us to do this with a minimal amount of coding effort.

1. INTRODUCTION

Legacy systems embody substantial institutional knowledge, which includes basic and refined requirements, design decisions, and invaluable advice and suggestions from domain users that have been implemented over the years. To effectively use these assets, it is important to employ a systematic strategy for continued evolution of the current system to meet the ever-changing mission, technology and user needs. Re-engineering has frequently been proven to be more cost effective than new development and is also known to better promote continuous software evolution.

However, the institutional knowledge implicit in a legacy system is difficult to recover after many years of operation, evolution, and personnel change. These software systems were originally written twenty or more years ago using what many now view archaic and ad-hoc methods. Such legacy systems usually lack accurate documentation, modular structure, and coherent abstractions that correspond to current or projected requirements. Past optimizations and design changes have spread design decisions that now must be changed over large areas of the code, and may have introduced inconsistencies and faults.

Software re-engineering can be defined as the systematic transformation of an existing system into a new form to realize quality improvements, such as increased or enhanced functionality, better maintainability, configurability, reusability, performance, or evolvability at a reduced cost, schedule, or risk to the customer. This process involves recovering existing software artifacts from the system and then transforming and re-organizing them as a basis for future evolution of the system. Since typical legacy systems were originally designed and implemented using a functionally based approach, some transformation of the recovered information is necessary in order to obtain an object-oriented model. It is often very difficult to obtain a transformed specification that accurately represents the desired requirements.

Since legacy systems are usually re-engineered only when the existing systems need some kind of improvement, it is unlikely that the initial version of the reconstructed requirements adequately reflects current user needs. Prototyping provides a means to identify and validate changes to system requirements while simultaneously enabling prospective users to get a feel for new aspects of the proposed system. It is a well-established approach that can be highly effective in increasing software quality [15]. When used in conjunction with conducting a major re-engineering effort, prototyping can be extremely useful in assisting in many areas of software modification, validation, risk reduction, and the refinement of new software architectures and user requirements.

This paper describes a case study that illustrates the effective use of computer-aided prototyping techniques for re-engineering legacy software [3, 16]. The case study consists of developing an object-oriented modular architecture for the existing US Army Janus(A) combat simulation system [19], and validating the architecture via an executable prototype using the Computer Aided Prototyping System (CAPS), a research tool developed at the Naval Postgraduate School [14]. Janus(A) is a software-based war game that simulates ground battles between up to six adversaries [9]. It is an interactive, closed, stochastic, ground combat simulation with color graphics. Janus is "interactive" in that command and control functions are entered by military analysts who decide what to do in crucial situations during simulated combat. The current version of Janus operates on a Hewlett Packard workstation and consists of over 350,000 lines of FORTRAN code.

The FORTRAN modules are organized as a flat structure and interconnected with one another via 129 FORTRAN COMMON blocks, resulting in a software structure that makes modification to Janus very costly and error-prone. The Software Engineering group at the Naval Postgraduate School was tasked to extract the existing functionality through reverse engineering and to create a base-line object-oriented architecture that supports existing and required enhancements to Janus functionality.

The paper presents the re-architecturing process and the resultant object-oriented architecture in Sections 2 and 3. Section 4 describes the use of computer aided prototyping to validate the resultant architecture and Section 5 draws some conclusions.

2. REVERSE ENGINEERING

The re-architecturing process used in the case study consists of 3 major phases: reverse engineering, object-oriented design and design validation via prototyping (Figure 1).

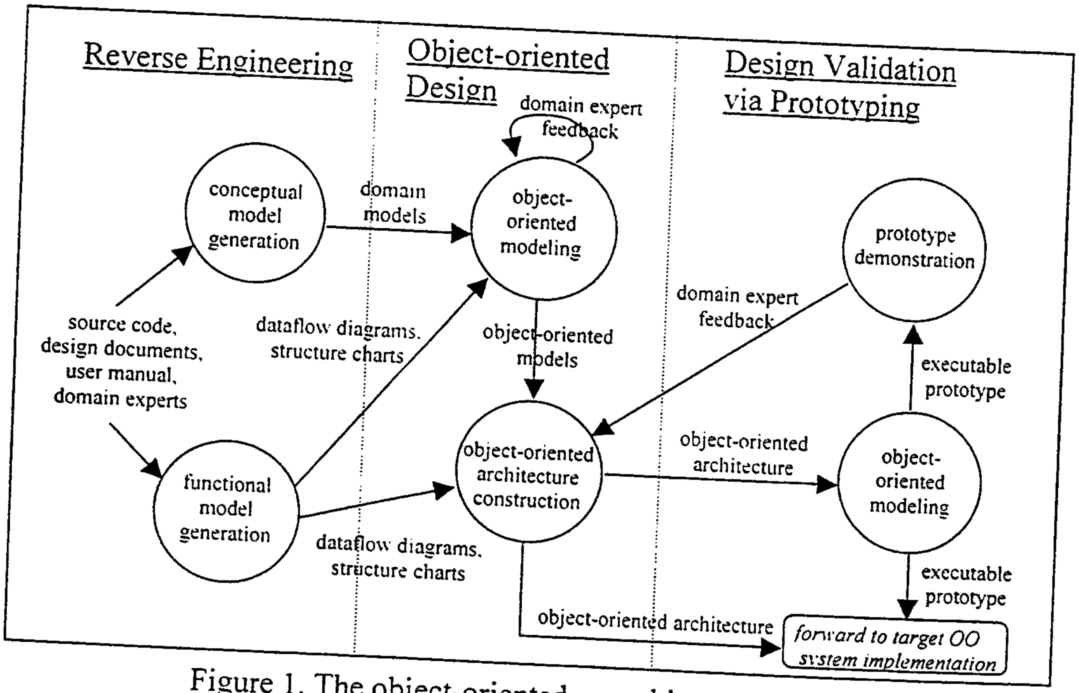

Reverse Engineering
Object-oriented Design
Design Validation via Prototyping
domain expert feedback
conceptual model generation
domain models
object-oriented modeling
prototype demonstration
source code, design documents, user manual, domain experts
dataflow diagrams, structure charts
object-oriented models
domain expert feedback
executable prototype
functional model generation
object-oriented architecture construction
object-oriented architecture
object-oriented modeling
dataflow diagrams, structure charts
executable prototype
object-oriented architecture
forward to target OO system implementation

Figure 1. The object-oriented re-architecturing process.

The first phase is reverse engineering. Input to this phase includes the legacy source code, design documents, user manuals, and information from domain experts. Since the goal of the initial re-engineering effort is to duplicate the functionality of the existing system within a modular, extensible architecture and to reuse domain concepts, models and algorithms instead of the existing code, we should avoid including any requirements/constraints that are consequences of issues related to FORTRAN implementation. The best places to extract domain concepts from the existing system are the user manuals and the database management system manuals. These manuals were written using the lingo of the user community and should be relatively free of implementation details. We found the JANUS Data Base Management Program Manual [10] particularly useful because it contains detailed information on what kind of data are needed to model the battlefield and how they are organized (logically) in the database. The top-level structure of the database is shown in Figure 2.

Not shown in Figure 2 are the interdependencies between the data, whereby data entered in one category affect directly or indirectly the data in other categories. For example, the barrier delay attributes of the Engineer Data depend on specific weather conditions derived from the Weather Data and system functional characteristics derived from the System Data. The overall network of interdependencies is highly complex and can only be understood through construction and analysis of a functional model of the existing Janus software.

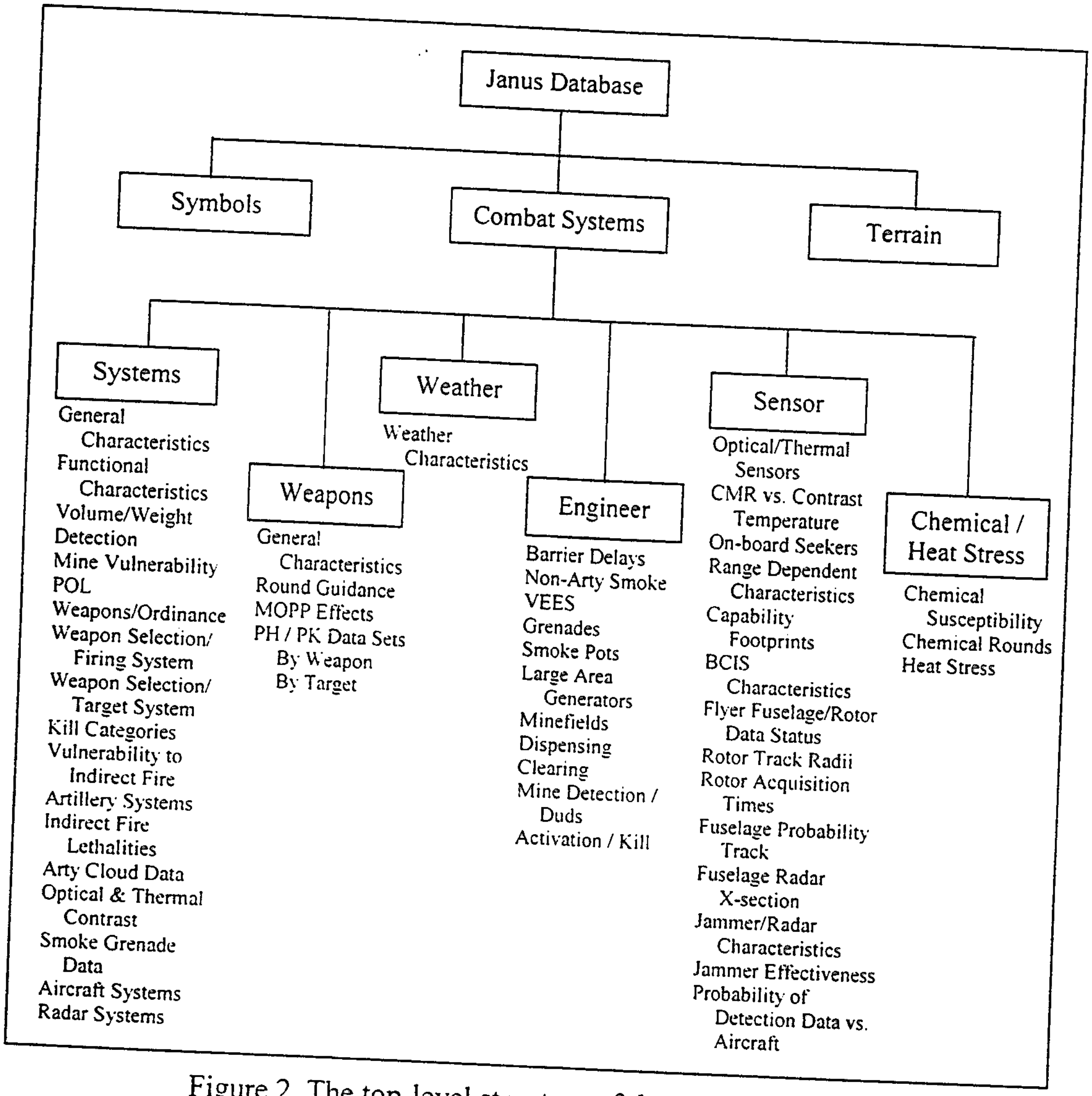


Figure 2. The top-level structure of the Janus Database.

Analysis of the legacy implementation is a daunting but inescapable part of this step. If printed out at 60 lines per page, 350,000 lines would fill almost 6000 pages. We recoiled from the magnitude of this effort and analyzed the Janus User's manual [9], the Janus Programmer's Manual [7], the Janus Software Design Manual [8], and the Janus Algorithm Document [18] instead. These documents helped us get started because they contained higher level information and were much shorter than the code. However, they were also older, and it was a constant struggle to determine which parts were still accurate, and which were not. In hindsight, avoiding analysis of the code was a mistake that slipped the schedule of the project by several months. Understanding a design of this complexity requires time for mental digestion, even with tool support and judicious sampling. We should have started analysis of the code right away and should have persistently continued this task in parallel with all other re-engineering activities. Cross-fertilization between all the tasks would have helped us recognize some dead-end directions earlier and would have enabled us to spend meeting time more effectively.

Using manual techniques augmented with simple UNIX shell commands, we were able to walk through the code and get a fairly good idea of what each subroutine was designed to do. We also used the Software Programmers' Manual [7] to aid in understanding each subroutine's function. In doing so we were able to group the subroutines by functionality to get a better understanding of the major data flows between programs and develop functional models from the data flows. We used CAPS to assist in developing the abstract models. CAPS allowed us to rapidly graph the gathered data and transform it into a more readable and usable format. Additionally, CAPS enabled us to concurrently develop our diagrams, and then join them together under the CAPS environment, where they can be used to generate an executable model.

We also had a series of brief meetings with the client, TRAC-Monterey, asking questions and making notes on the system's operation and its current functionality. We paid attention to the client's view of the system to gather their ideas on its strengths, weaknesses, and desired and undesired functionality. These meetings were indispensable because they gave us information that was not present in the code. Since we were not familiar with the domain of ground combat simulation, we were using these meetings to determine the requirements of this domain, often playing the role of "smart ignoramuses" [4]. Domain analysis has been identified as an effective technique for software re-engineering [17]. Our experience suggests that competent engineers unfamiliar with the application domain have an essential role in re-engineering as well as in requirements elicitation because lack of inessential information about the application domain makes it easier to find new, simpler design structures and architectural concepts to guide the re-engineering effort.

3. OBJECT-ORIENTED DESIGN

Next, we developed object models and architecture of the Janus System using the aforementioned materials and products, to create the modules and associations amongst them. Information modeling is needed to support effective re-engineering of complex systems [5]. This was probably the most difficult and most important phase. It required a great deal of analysis and focus to transform the currently scattered sets of data and functions into small, coherent and realizable objects, each with its own attributes and operations. In performing this phase, we used our knowledge of object-oriented analysis and applied the OMT techniques [20] and the UML notations to create the classes and associated attributes and operations [21]. This was a crucial phase because we had to ensure that the classes we created accurately represented the functions and procedures currently in the software.

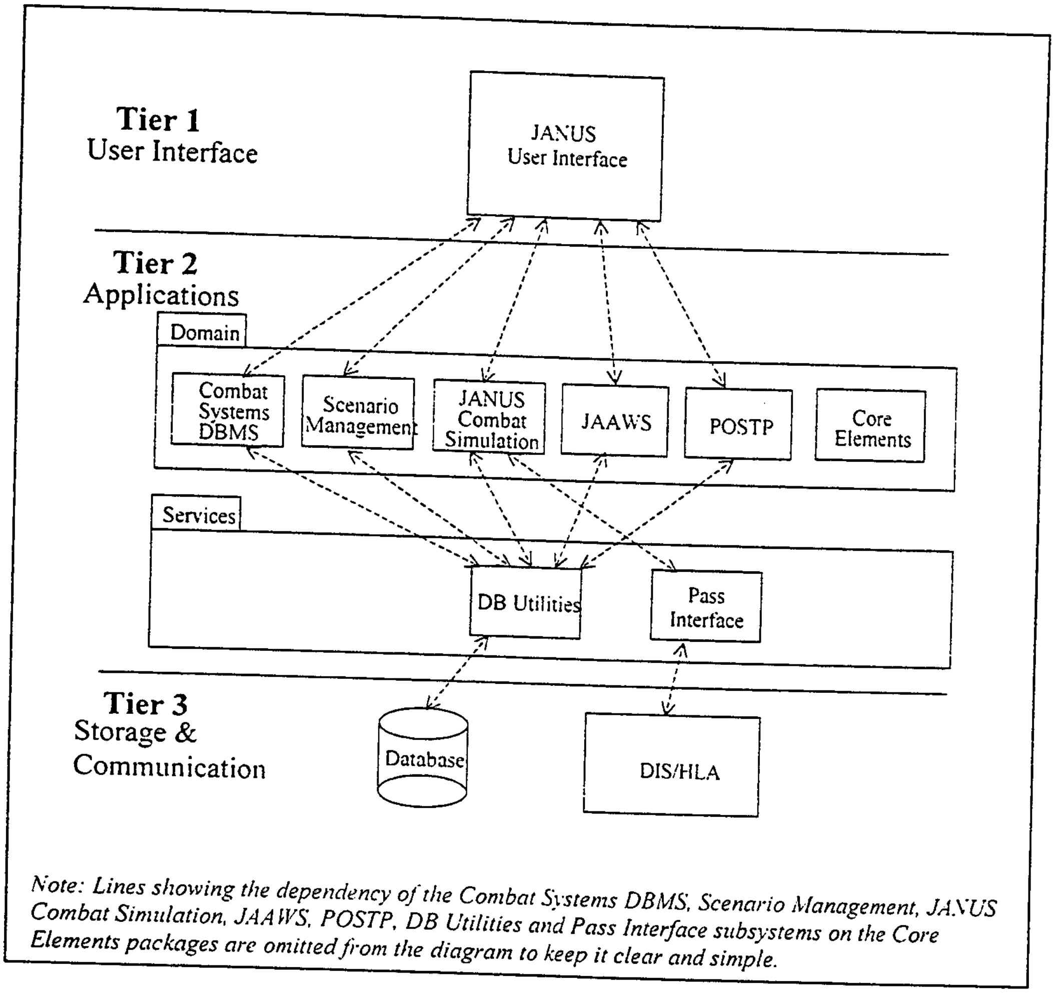
Restructuring software to identify data abstractions is a difficult part of the process. Transformations for meaning-preserving restructuring can be useful if tool support is available [6]. We used the HP-UNIX systems at the TRAC-Monterey facility to run the

Janus simulation software to aid in verifying and supplementing the information we obtained from reviewing the source code and documentation. This step enabled us to better analyze the simulation system, gaining insight into its functionality and further concentrate on module definition and refinement.

The re-engineering team met several times each week for a period of two and a half months to discuss the object models for the Janus core data elements and the object-oriented architecture for the Janus System. We presented the findings to the Janus domain experts at least once per week to get feedback on the models and architectures being constructed. In addition, the re-engineering team also presented the findings to members of the OneSAF project, the Combat21 project, and the National Simulation Center project. We found that information from these domain experts was essential for understanding the system, particularly in cases where the legacy code did not correspond to stakeholder needs. This supports the hypothesis advanced in [11] that the involvement of domain experts is critical for nontrivial re-engineering tasks.

Early involvement of the stakeholders in the simulation community also paid off in the long run. Both the National Simulation Center and Combat21 projects were able to save time and money by reusing our work and came up with designs that look remarkably like ours (although much larger). Now. OneSAF developers have been directed to look at the Combat21 class design and reuse as much as possible. So, our efforts have directly benefited other simulation developers.

Based on the feedback from the domain experts, the re-engineering team revised the object models for the Janus core elements and developed a 3-tier object-oriented architecture for the Janus System (Figure 3).



Note: Lines showing the dependency of the Combat Systems DBMS, Scenario Management, JANUS Combat Simulation, JAAWS, POSTP, DB Utilities and Pass Interface subsystems on the Core Elements packages are omitted from the diagram to keep it clear and simple.

Figure 3. The proposed 3-tier object-oriented architecture.

We extracted most of the data and operations from the existing Combat System DBMS, Scenario Management, Janus Combat Simulation, JAAWS and POSTP subsystems and encapsulated them as simulation objects in the Core Elements package, leaving only application specific control codes that use the simulation objects in each of these five subsystems. Figures 4 and 5 show the top level class structures of the object models of the core elements. Details of the associated attributes and operations can be found in [2, 23] and are omitted from these diagrams due to space limitations.

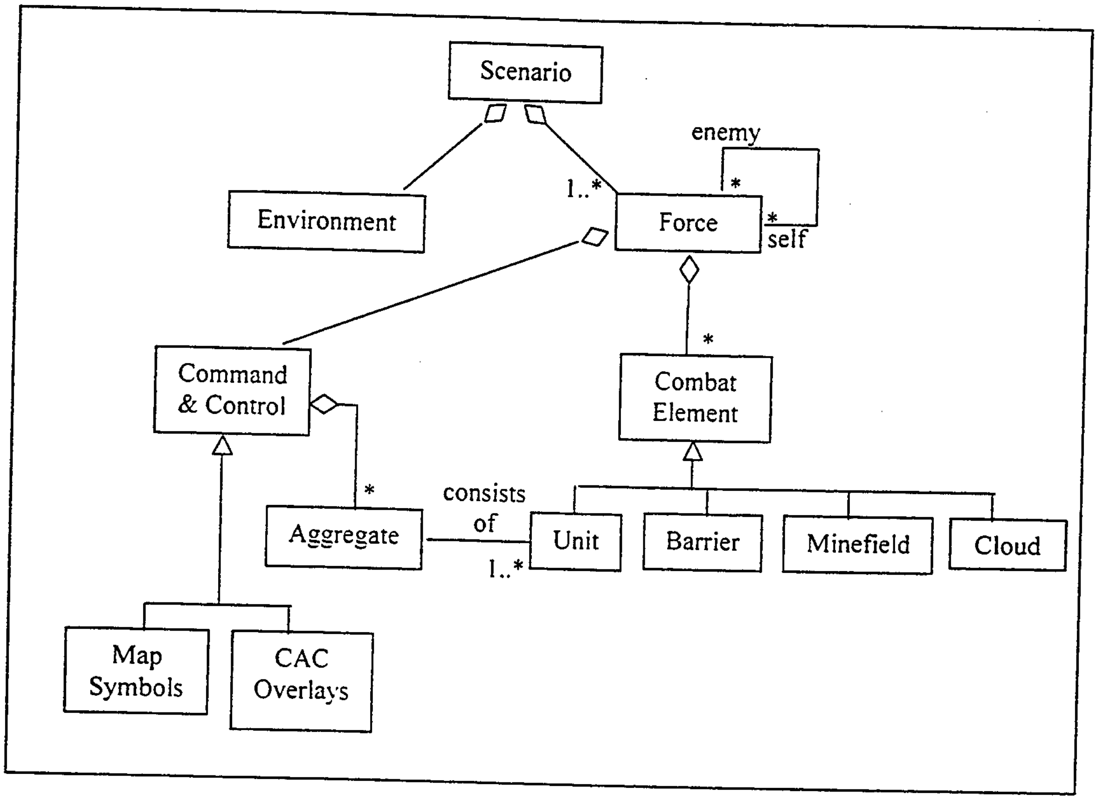


Figure 4. The top-level structure of the Janus Core Elements Object Model.

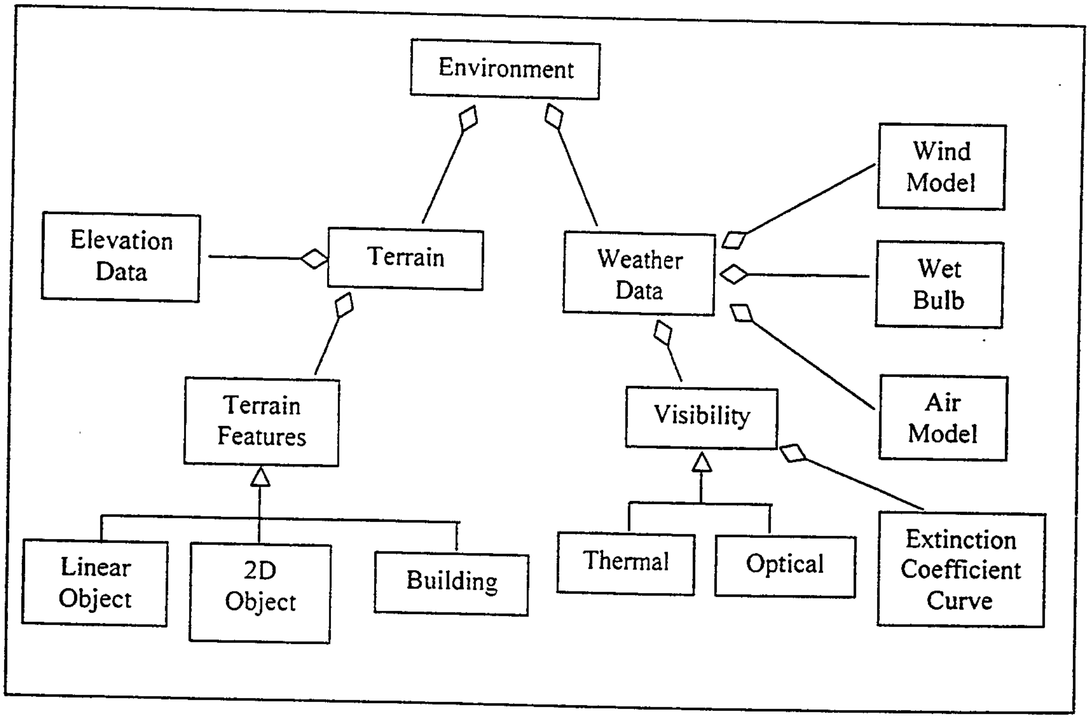


Figure 5. The Environment Object Class.

Central to the Janus Combat Simulation Subsystem is the program RUNJAN, which is the main event scheduler for the simulation. RUNJAN determines the next scheduled event and executes that event. If the next scheduled event is a simulation event, RUNJAN will advance the game clock to the scheduled time of the event and perform that event. The existing Janus Simulation System uses 17 different categories to characterize the events. RUNJAN then handles these 17 events using the following event handlers:

1) DOPLAN - Interactive Command and Control activities

2) MOVEMENT - Update unit positions

3) DOCLOUD - Create and update smoke and dust clouds

4) STATEWT - Periodic activity to write unit status to disk

5) RELOAD - Plan and execute the direct fire events

6) INTACT - Update the graphics displays

7) CNTRBAT - Detect artillery fire

8) SEARCH - Update target acquisitions, choose weapons against potential targets, and schedule potential direct fire events

9) DOCHEM - Create chemical clouds and transition units to different chemical states

10) FIRING - Evaluate direct fire round impacting and execute indirect fire missions

11) IMPACT - Evaluate and update the results of an indirect round impacting

12) RADAR - Update an air defense radar state and schedule direct fire events for "normal" radar

13) COPTER - Update helicopter states

14) DOARTY - Schedule indirect fire missions

15) DOHEAT - Update unit's heat status

16) DOCKPT - Activity to record automatic checkpoints

17) ENDJAN - Housekeeping activity to end the simulation

The existing event scheduler uses global arrays and matrices to maintain the attributes of the objects in the simulation. Hence, one of the major tasks in designing an object-oriented architecture for the Janus Combat Simulation Subsystem was to distribute the event handling functions to individual objects. However, many of the current event handler categories contained redundant code. They did not seem to be independent of each other and were not consistent with the class hierarchy we created. For example, the set of event handlers used to simulate the activities of a particular unit to search for targets, select weapons, prepare for a direct fire engagement, and then execute that direct fire engagement differs depending upon whether the unit has a normal radar, special radar, or no radar at all. The existing Janus Simulation System uses the RADAR event handler to carry out the entire procedure if the unit has normal radar. However, it uses the SEARCH, RADAR, and RELOAD event handlers to carry out the procedure if the unit has special radar. Finally the system uses the SEARCH and RELOAD event handlers to conduct the procedure if the unit has no radar at all. We conjecture that this lack of uniformity is due to a series of software modifications made by different people at different times without full knowledge of the software structure. The example also illustrates another problem: the legacy event handlers were not designed to perform independent tasks, and had complicated interactions with each other.

It was necessary to redefine some event categories in order to reduce interdependencies between the event handlers, to factor simulation behavior into more coherent modules, to eliminate redundant coding of the same or similar functions and to take advantage of dynamic dispatching of event handling functions in the object-oriented architecture. Moreover, the Janus system was originally designed to work in isolation, and has since been adapted to interact with other simulation systems. Interactions between the simulation engine and the world modeler (the interface to the distributed simulation network) are performed implicitly within the various event handlers in the existing Janus. Such interactions are made explicit in the new architecture in order to provide a uniform framework to update World Model objects during the simulation.

The new architecture uses an explicit priority queue of event objects to schedule the simulation events. We were able to reduce the total number of event handlers needed in the simulation, from 17 to 14, by eliminating identified redundant code (Figure 6). The 14 remaining event handlers are as follows:

1) DOPLAN - Interactive Command and Control activities

2) MOVE_UPDATE_OBJ – Move and update the objects in the simulation

3) SEARCH – Search for potential targets based on the detection devices available to the objects

4) CHOOSE_DIRECT_FIRE_TARGETS – Once search is complete, choose best target to engage. In future simulations, implementations may allow users to choose targets

5) COUNTERBATTERY – Simulate counter battery radar to detect artillery fire

6) DO_DIRECT_FIRE – Execute direct fire events and update ammunition status

7) DO_INDIRECT_FIRE – Execute indirect fire events and update ammunition status

8) IMPACT_EFFECTS – Calculate results of round impacting

9) UPDATE_HEAT_STATUS – Update unit's heat status

10) UPDATE_CHEMICAL_STATUS – Update unit's chemical status

11) DISPLAY – Update the graphics display

12) WRITE_STATUS – Periodic activity to write units status to disk

13) CHECK_POINT – Activity to record automatic checkpoints

14) END_SIMULATION – Activity to end the simulation

We tried to make the actions of the new event handlers independent and orthogonal. Independent means that one event handler does not invoke or depend on the action of another. Orthogonal means that the purpose of one event handler is completely separate from that of another. Although our architecture does not completely meet these goals, it comes much closer to them than the legacy design does. We believe that these properties of the architecture are desirable because they impose a partitioned structure on the system that aids future enhancements and modifications. If an enhancement affects only one kind of event, then it becomes relatively easy to isolate the affected part of the code. If suitable naming conventions are followed, relatively low-tech tool support will be adequate for helping system maintainers find the parts of the code that must be understood and modified to make a future change to the system.

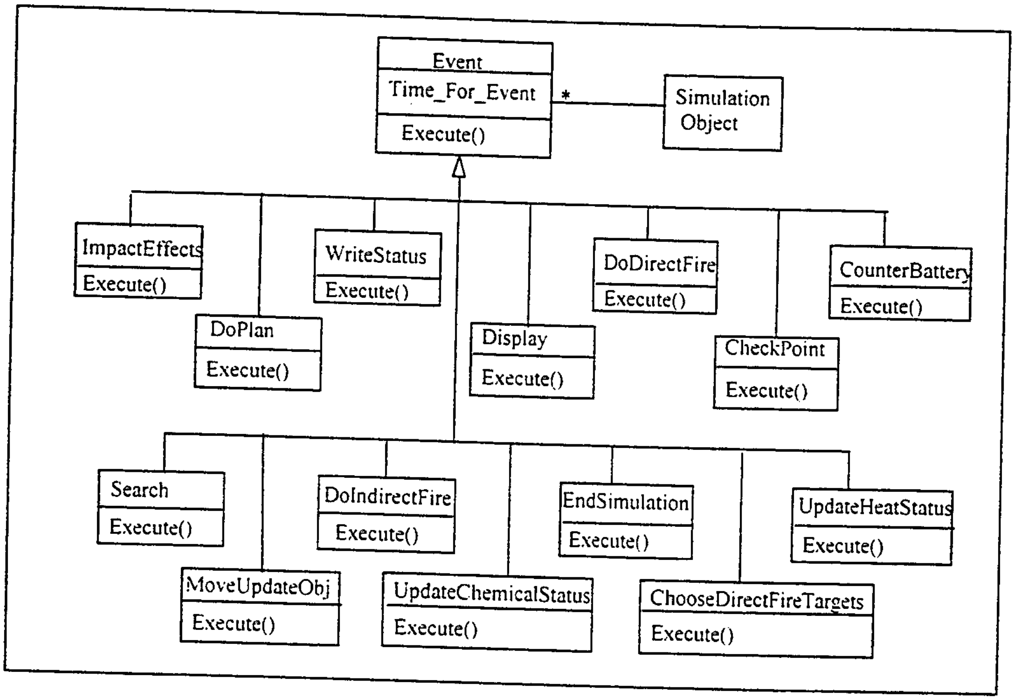


Figure 6. The event class hierarchy.

Every event has an associated simulation object in the new architecture. This associated object is the target of the event. Depending on the subclass to which an event object belongs, the "execute" method of the event will invoke the corresponding event handler of the associated simulation object (Figure 7). The simulation object superclass defines the interface of the event handlers for the event groups. At the highest level, it provides an empty body as the default implementation for the event handlers. Events are dispatched to the appropriate subclass. If there is something more specific that needs to be done for instances of the subclass, the event handler of the subclass overrides the inherited method in order to simulate the desired behavior.

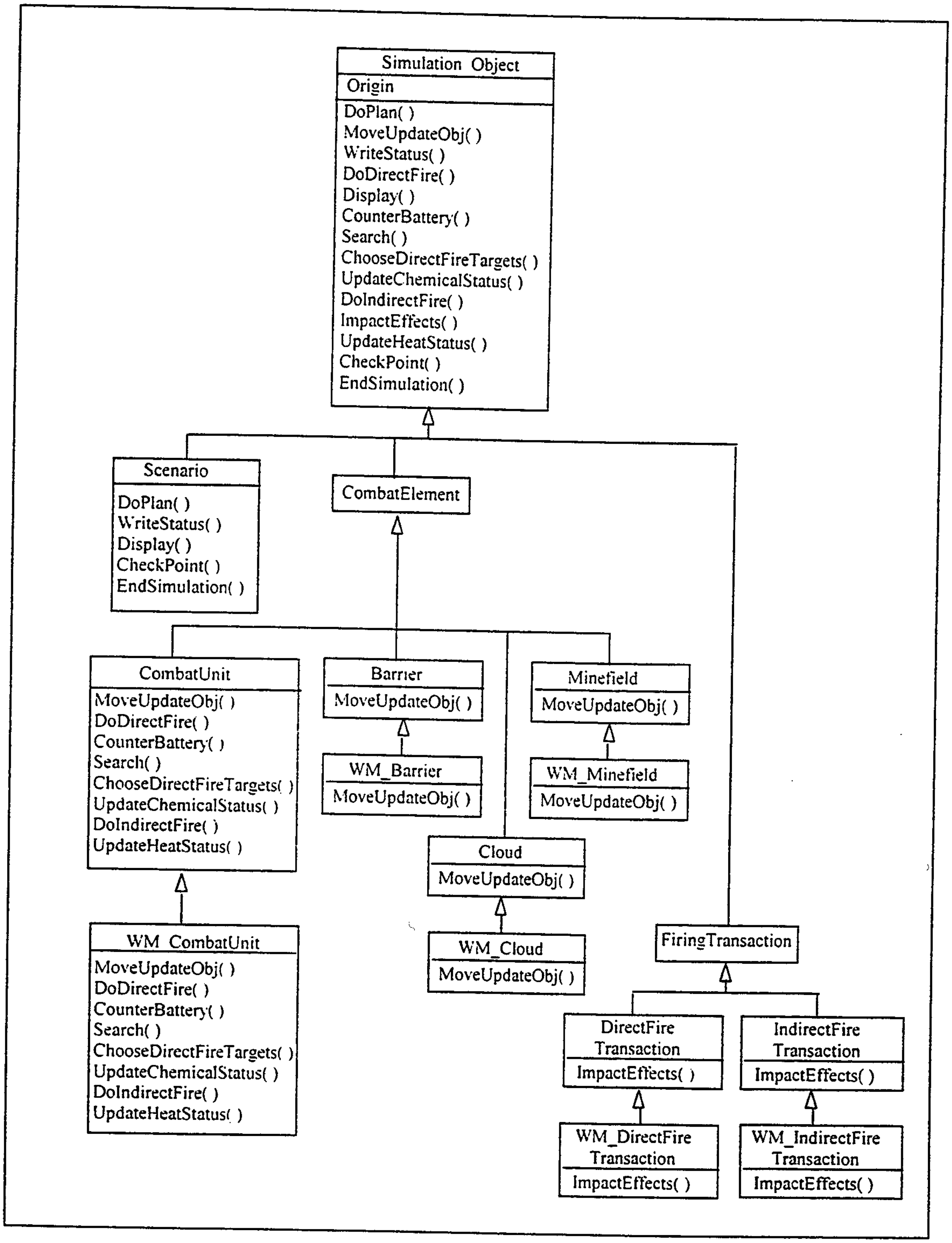


Figure 7. The simulation object class hierarchy.

The above architecture enables a very simple realization of the main simulation loop:

```
initialization;
while not_empty(event_queue) loop
    e := remove_event(event_queue);
    e.execute();
end loop;
finalization;
```

Note that this same code is used to handle all of the event handlers, including those for future extensions that have not yet been designed. Event objects with associated simulation objects are created and inserted into the event queue by the initialization procedure, the constructors of simulation objects, and the actions of other event handlers. Depending on the actual event, events are inserted into an event priority queue based on time and priority.

Our newly designed architecture eliminates the need for the simulation loop to know what kind of object it is handling. Thus when adding an object type not yet designed, the simulation loop does not require additional code to invoke the new object's event handlers. By localizing all changes to the newly added object class, our architecture eliminates the possibility of introducing errors into the existing parts of the simulation.

4. DESIGN VALIDATION VIA PROTOTYPING

The process of transforming a design developed using the functional approach into an object-oriented design introduces risks of unintentionally altering system behavior. In the context of our case study, the resultant object oriented architecture and the new event dispatching control structure are areas of high risk since they differ significantly from the functional design of the legacy software. UML provides two ways to model behavior. One is to capture the behavior of individual objects over time using state machines, and the other is to capture the interactions of a set of objects in the system using sequence diagrams and collaboration diagrams. While state machines are precise, they only focus on a single object at a time and is hard to understand the behavior of the system as a whole. The sequence diagrams and the collaboration diagrams, on the other hand, lack a formal semantics for precise description of the system behaviors.

One way to reduce the risk is to validate the dynamic behavior of the proposed architecture and to refine the interfaces of subsystems via prototyping at the early design stage. To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. Computer aid for constructing and modifying prototypes makes this feasible [15]. The CAPS system is an integrated set of software tools that generate source programs directly from high-level requirement specifications.

Due to time and resource limitations, we developed a prototype for only a very small simulation run, which consists of a single object (a tank) moving on a two-dimensional plane, three event subclasses (*move*, *do_plan*, and *end_simulation*), and one kind of post-processing statistics (fuel consumption).

We developed an executable prototype using CAPS. Figure 8 shows the top-level structure of the prototype, which has four subsystems: *janus*, *gui*, *jaaws* and the *post_processor*. Among these four subsystems, the *janus* and the *gui* subsystems (depicted as double circles) are made up of sub-modules as shown in Figures 9 and 10, while the *jaaws* and the *post_processor* subsystems (depicted as single circles) are mapped directly to modules in the target language. After entering the prototype design into CAPS, we used the CAPS execution support system to generate the code that interconnects and controls these subsystems. In addition, a simple user interface was developed using CAPS/TAE [22] (Figure 11). The resultant prototype has over 6000 lines of program source code, most of which was automatically generated, and contains enough features to exercise all parts of the architecture. The code that handles the motion of a generic simulation object was very simple, but it was designed so that it would work in both two and three dimensions without modification (currently the initialization and the movement plan of the tank object never call for any vertical motion). The code was also designed to be polymorphic. just as was the main event loop. This means the same code will handle the motion of all kinds of simulation objects without any modifications, including new types of simulation objects that are part of currently unknown future enhancements to Janus and have not yet been designed or implemented.

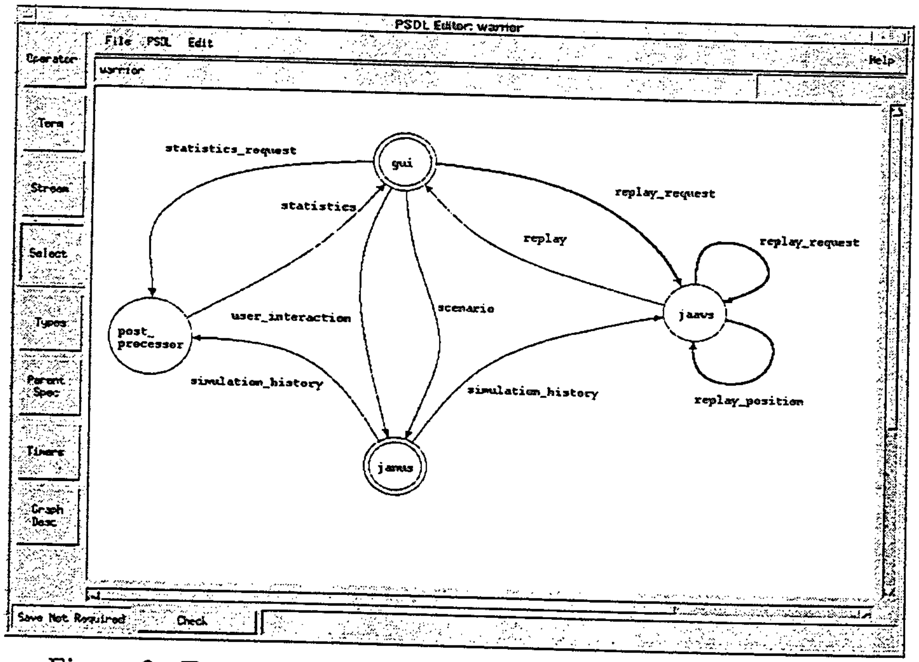


Figure 8. Top-level decomposition of the executable prototype.

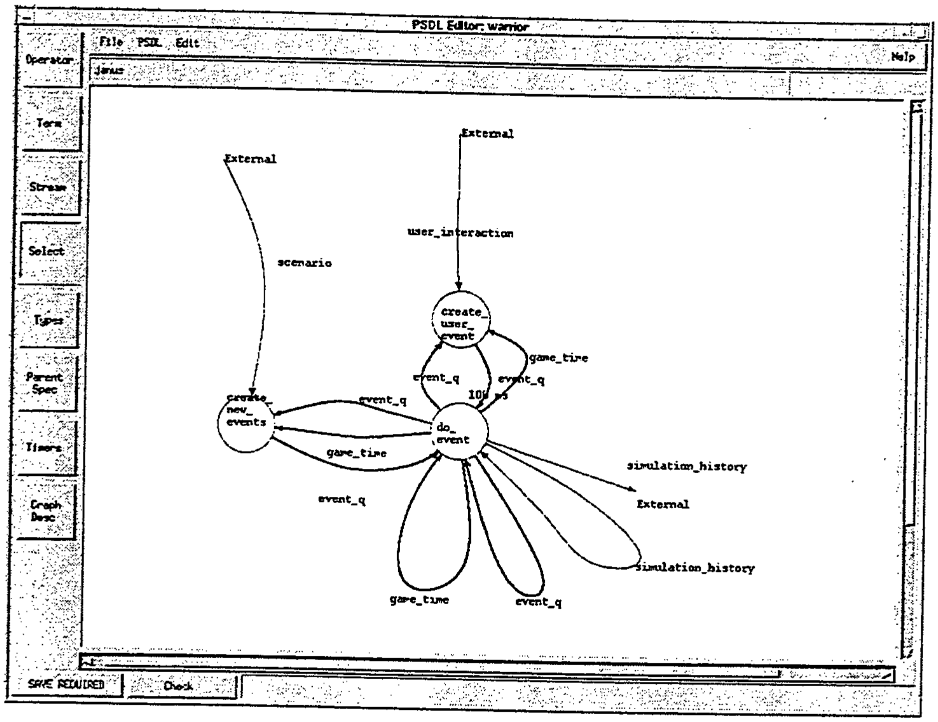


Figure 9. The JANUS subsystem of the executable prototype.

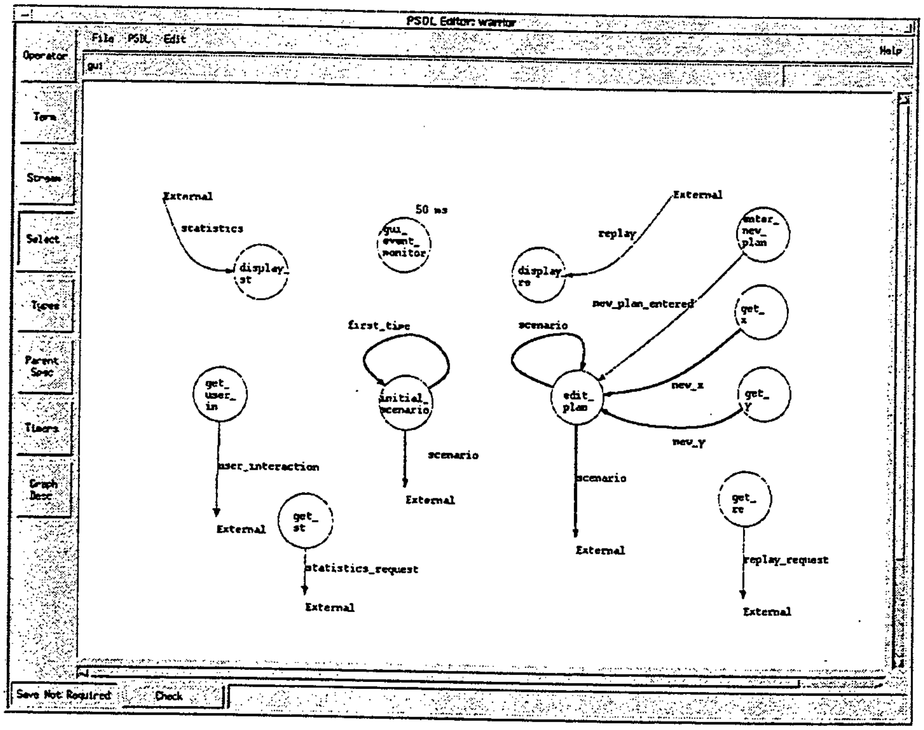
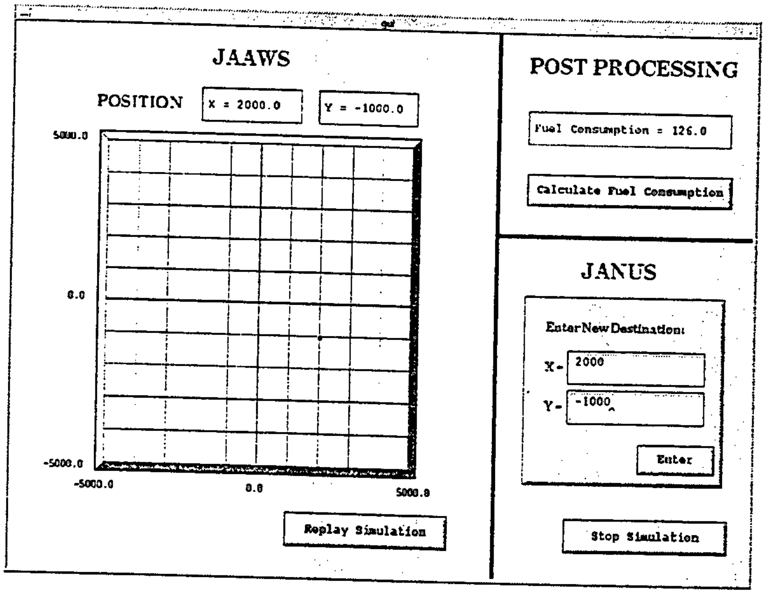


Figure 10. The GUI subsystem of the executable prototype.



POSITION X = 2000.0 Y = -1000.0

Fuel Consumption = 126.0

EnterNewDestination:

X- 2000

Y- -1000

Figure 11. The Graphical User Interface of the executable prototype

Our prototyping experiment showed that the proposed object-oriented architecture allows design issues to be localized and provides easy means for future extensions. We started out with a prototype consisting of only two event subclasses (*move* and *end_simulation*) and were able to add a third event subclass (*do_plan*) to the prototype without modifying the event control loop of the Janus combat simulator.

We also demonstrated the use of inheritance and polymorphism to efficiently extend/specialize the behavior of combat units. For example, the *move_update_object* method of a tank subclass uses the general-purpose method from its superclass to compute its distance traveled and a specialized algorithm to compute its fuel consumption. We simply include one statement to invoke the *move_update_object* method of its superclass followed by three lines of code to update its fuel consumption. Moreover, other combat unit subclasses can be added easily to the prototype without the need to modify the event scheduling/dispatching code and usually without modifying existing event handlers.

The issues raised by the design of the prototype also resulted in the following refinements to the proposed architecture:

1. Extend the interface of the *Execute_Event* operation to return the time at which the next event is to be scheduled for the same simulation object, and introduce a special time value "NEVER" to indicate that no next event is needed. The proposed change turns the communication between the event dispatcher and the simulation objects from a peer-to-peer communication into a client-server communication. This change eliminates dependencies of event handlers on event queue details and allows the event dispatcher to use a single statement to schedule all recurring events for all event types.

2. Instead of recording the history of a simulation run in sets of data files, model the simulation history as a sequence of events. The proposed change provides a simple and uniform way to handle history records for all events, and allows the same modular architecture to be used for real-time simulations as well as post-simulation analysis. It also eliminates the need for the write-status event, reducing the number of events still further. This approach provides the greatest possible resolution for the event histories, which implies that any quantity that could have been calculated during the simulation can also be calculated by a post-simulation analysis of the event history, without any loss of accuracy. The only constraint imposed by this design refinement is that the simulation objects in the events must be copied before being included in the simulation history, to protect them from further changes of state as the

simulation proceeds. This constraint is easy to meet in a full-scale implementation because the process of writing the contents of an event object to a history file will implicitly make the required copy.

The prototyping effort also exposed a design issue - should null events appear in the event queue? A null event is one that does not affect the state of the simulation, such as a move event for an object that is currently stationary. The prototype version adopted the position that such events should not be put in the event queue, since this corresponds to current scheduling policies in Janus, and appears at first glance to improve efficiency.

Our experience with the development of the prototype suggests that this decision complicates the logic and may not in fact improve efficiency. In particular, the process *create_new_events* (see Figure 9) could be eliminated if we allowed null events. This process scans all simulation objects once per simulation cycle to determine if any dormant objects have become active, and if so, schedules events to handle their new activity. The alternative is to have the constructor of each kind of simulation object schedule all of its initial events, and to have each event handler specify the time of next instance of the same event even if there is nothing for it to do currently. Handlers might still set the time of its next event to NEVER in the case of a catastrophic kill; however this is reasonable only if it is impossible to repair or restore the operation of the units that have suffered a catastrophic kill. The reasons why this design change may improve efficiency in addition to simplifying the code are that:

1. the check for whether a dormant object has become active is done less often - once per activity of that object, rather than once per simulation cycle,

2. executing a null event is very fast - a few instructions at most, so the “unnecessary” null events will not have much impact on execution time, and
3. the computation to find and test all simulation objects periodically would be eliminated.

We recommend allowing null events in the event queue, and explicitly scheduling every kind of event for every object unless it is known that there cannot be any non-empty events of that type in any possible future state of the object. For example, under the proposed scheduling policy, immobile or irrecoverably damaged objects would not need to schedule future move events, but those that are currently at their planned positions would need to do so, because a change of plan could cause them to move again in the future, even though they are not currently moving. The resulting architecture enables a very simple realization of the main simulation.

5. CONCLUSION

Our conclusion is that substantial and useful computer aid for re-engineering is possible at the current state of the art. Human analysts and domain experts must also play an important part of the process because much of the information needed to do a good job is not present in the software artifacts to be re-engineered. Success depends on cooperation between skilled people and appropriate software tools.

The missing information needed for re-engineering is related to deficiencies of the current system at all levels, from requirements through design and implementation. Thorough and accurate knowledge of these deficiencies is crucial for success. The clients never want the re-engineered system to have the exactly same behavior as the legacy system - if they were satisfied, there would be little motivation to spend time, effort, and resources on a re-engineering project. Even if a system is being re-engineered for the ostensible goal of porting to different hardware, the desired behavior at the interface to the hardware and systems software will be different.

In practical situations, the requirements for the re-engineered system are different from those for the legacy system. Key parts of the requirements for the new system are often missing or incorrect in the legacy documents. Some of that information is present only in the minds of the clients, often fragmented and scattered across members of many different organizations. Communication is a large part of the process, and that communication cannot be automated away, although it can be enhanced by appropriate use of prototyping. We found that the most important communications were those regarding newly recognized requirements issues, and that such recognition were often triggered by discussions between people with different areas of expertise.

Uncertainties about the true requirements play a central role in both re-engineering and the development of new systems. We therefore hypothesized that prototyping could play a valuable role in re-engineering efforts. Our experience in the case study reported here support that hypothesis.

We also found that prototyping can contribute substantially to the process of inventing, correcting, and refining the conceptual structures on which the architecture of the new system will be based. Most legacy systems are too complicated for individuals to understand.

This maze of details hides potential opportunities for simplifying and regularizing the conceptual structure of the system to be re-engineered, and makes it difficult to recognize deficiencies in design and architectural structure. The amplification process implicit in constructing skeletal prototypes helps expose such opportunities.

We found that there are fundamental conceptual errors embodied in the legacy structures and algorithms. Some of those errors were exposed when structural asymmetries and irregularities are discovered in the process of extracting a model of the legacy software. Others were discovered only with the help of the oversimplified models that are common in the early stages of prototyping a proposed new architecture. Constructing a small and simple instance of the proposed architecture raises many of the main design issues, and the simplicity of the model makes it much easier to consider and evaluate alternative designs to find improved structures.

To be effective, prototypes must be constructed and modified rapidly, accurately, and cheaply. The UML interaction diagrams lack the preciseness to support automatic code generation for the executable prototype. This weakness can be remedied by the use of the prototype language PSDL [12, 13] and the CAPS prototyping environment, which provide effective means to model the system's dynamic behavior in a form that can be easily validated by user via prototype demonstration.

ACKNOWLEDGEMENT

The authors thank Dr. David Hislop, COL Michael McGinnis, MAJ Gerald Pearman, MAJ LeRoy Jackson, MAJ William Murphy, SFC Cary Augustine, Harold Yamauchi and Bill Caldwell for their help and support for the project. This research was supported in part by the U.S. Army Research Office under contract # 35037-MA and in part by the U. S. Army Training and Doctrine Analysis Command.

REFERENCES

[1] I. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering," *Proceeding of the 4th Workshop on Reverse Engineering*, IEEE Computer Society, 1997, pp. 104-113.

[2] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, *Re-engineering the Janus(A) Combat Simulation System*. Technical Report NPS-CS-99-004, Computer Science Department, Naval Postgraduate School, Monterey, CA, January 1999.

[3] V. Berzins, M. Shing, Luqi, M. Saluto and J. Williams, "Architectural Re-engineering of Janus using Object Modeling and Rapid Prototyping," to be published in the journal *Design Automation for Embedded Systems*. A preliminary version of the paper also appeared in *Proceedings of the 10th IEEE International Workshop in Rapid Systems Prototyping*, Clearwater Beach, Florida, 16-18 June 1999, pp. 216-221.

[4] D. Berry, Formal Methods: The Very Idea, "Some Thoughts About Why They Work When They Work," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems,* 1998, pp. 9-18.

[5] O. Bray and M. Hess, "Reengineering a Configuration-Management System," *IEEE Software*, Vol. 12, No. 1, Jan. 1995, pp. 55-63.

[6] V. Cabaniss, B. Nguyen and J. Moregenthaler, "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems," *IEEE TSE,* Vol. 24, No. 7, July 1998, pp. 534-558.

[7] *Janus 3.X/UNIX Software Programmer's Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[8] *Janus 3.X/UNIX Software Design Manual*, Prepared for: Headquarters TRADOC Analysis Center, Ft. Leavenworth, Kansas. Prepared by: Titan, Inc. Applications Group, Leavenworth, Kansas, Nov. 1993.

[9] *Janus Version 6 User's Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[10] *Janus Version 6 Data Base Management Program Manual*, Simulation, Training & Instrumentation Command, Orlando, Florida, 1995.

[11] S. Jarzabek and P.K. Tan, "Design of a Generic Reverse Engineering Assistant Tool," *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, 1995, pp. 61-70.

[12] B. Kraemer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Transactions on Software Engineering*, Vol. 19, No. 5, May 1993, pp. 453-477.

[13] Luqi, V. Berzins, and R. Yeh. "A Prototyping Language for Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 14, No.10, October 1988, pp. 1409-1423.

[14] Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, Vol. 5, No. 2, 1988, pp. 66-72.

[15] Luqi, "System Engineering and Computer-Aided Prototyping," *Journal of Systems Integration - Special Issue on Computer Aided Prototyping*, Vol. 6, No. 1, 1996, pp.15-17.

[16] Luqi, V. Berzins, M. Shing, M. Saluto, J. Williams, J. Guo and B. Shultes, "The Story of Re-engineering of 350,000 Lines of FORTRAN Code," *Proceedings of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Carmel, CA, 23-26 October 1998, pp. 151-160.

[17] M. Moore and S. Rugaber, "Domain Analysis for Transformational Reuse," *Proceedings of 4th Workshop on Reverse Engineering*, IEEE Computer Society, 1997, pp. 156-163.

[18] J. Pimper and L. Dobbs, *Janus Algorithm Document, Version 4.0*, Lawrence Livermore National Laboratory, California, 1988.

[19] L. Rieger and G. Pearman, "Re-engineering Legacy Simulations for HLA-Compliance," *Proceedings of the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*, Orlando, Florida, December 1999.

[20] J. Rumbaugh. M. Blaha, W. Premerlani. F. Eddy and W. Lorenzer, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[21] J. Rumbaugh. I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1999.

[22] *TAE Plus C Programmer's Manual (Version 5.1)*. Prepared for: NASA Goddard Space Flight Center. Greenbelt, Maryland. Prepared by: Century Computing, Inc., Laural, Maryland, April 1991.

[23] J. Williams and M. Saluto, *Re-engineering and Prototyping Legacy Software Systems-Janus Version 6.X*, master's thesis, Naval Postgraduate School, Dept. of Computer Science, Monterey, CA, March 1999.

Product Line Viewpoint and Validation Models

Nadar Nada, Luqi
Naval Postgraduate School
C.S. Dept. Code CS/ 833 Dyer Rd.
Monterey, CA. 93943 USA
+1 831 656 4075
nnada,luqi@cs.nps.navy.mil

Khaled Jaber
Case Western Reserve Univ.
C.S. Dept./10900 Euclid Ave.
Cleveland, OH. 44106 USA
+1 860 2149
jaber@lucent.com

David Rine
George Mason University
C.S. Dept. MS 4A5
Fairfax, VA 22030
+1 703 993 1546
drine@gmu.edu

ABSTRACT

A product line is a group of systems sharing a common, managed set of features that satisfy specific needs of a selected market or mission. In the product line approach, management, system developers, and a reuse team are interested in some views of the product line. In this paper a model is defined to present product lines, its derived products, and common assets used in these product lines. The model is used to convey views of interest to different stakeholders: management, system developers, and a reuse team in the product line approach. Its purpose is to capture information and present this information about organizations' product lines, and make it visible to the stakeholders inside and outside organizations. Management can use the model when producing new products of a product line, negotiating with customers, and assessing the benefits of adopting the product line approach. Product line developers can use the model when developing products of a product line. A reuse team can use the model through asset identifications, ensuring a successful use of asset base in and across product lines, and assessing the level of reuse.

Keywords
Product line, Product line architecture, COTS, Organizational components, Stakeholders, and System-unique components.

1 INTRODUCTION

Organizations that develop similar products are adopting the product line or product family approach to deploy systems faster, at a low cost, and a high quality. Systems are produced in a product line using common architecture and assets that are used across products. Organizations reuse common assets, integrated assets, etc. that would otherwise have to be needlessly repeated for each system.

Each stakeholder, i.e. management, systems developers, and reuse team is interested in a particular view of the product line. Management, for example, might be interested in viewing products of a product line to estimate time and schedules. Systems developers might be interested in a view of a product line looking for common assets. The reuse team might be interested in a view of a product line to assess the level of reuse in a product line. These are some of the interesting views.

We are presenting a product line viewpoint model that shows different views of the product line, its derived products, and common assets used. Also we are showing how the model conveys particular views interesting to management, systems developers, and reuse team.

Section 2 describes the product line concept. Section 3 describes the product line model. Section 4 describes views captured by the model. Section 5 is an empirical model for product line validation. Section 6 represents a repository support. Section 7 is the conclusion.

2 PRODUCT LINE CONCEPT

A product line is defined as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission [1, 4]. Products in the product line are engineered through customization from base requirements and standard product line architectures, and integration of common components rather than using system-unique software [2].

The product line architecture is one of the important assets shared by the systems in a product line. It provides the structure for building systems in the product line. All products are based on the product line architecture.

Product line assets are used across products in the product line. Product line assets depend on the solutions common to the products in a product line. Reusing these solutions reduces or eliminates work that otherwise would be required to build each product [3].

In the product line development, a dual life-cycle model can be used in which domain engineering is the process used to create domain artifacts useful across the entire

Table 1 The Viewpoint and Attribute Template.

Viewpoint Template

Reference	The viewpoint name
Attributes	Attributes providing view point information
Tasks	A reference to a set of event scenarios describing how viewers interact with the product line and their tasks
Sub-views	The names of sub-viewpoints

Attributes Template

View Entities	Attributes
Product line	Name, owner, intended market.
Product	Name, contact person, customer(s).
Product line architecture release	Contact person, release number. number of times reused. development time. number of staff. used architectural style. inter-component used communication mechanisms, operating systems(s)and platform(s).
Product release	Customers. release number. contact person. development time, development cost, when developed. number of staff, status, operating system(s) and platform(s).
COTS component release	Name. vendor. release number. contact person. cost, number of times reused, operating system(s) and platform(s).
Organizational component release	Name, release number. contact person. developed internally or externally, development cost, number of times reused. development time. number of staff, operating system(s) and platform(s).
System-unique component release	Name; release number. contact person. development cost, development time. and number of staff, operating system(s) and platform(s).

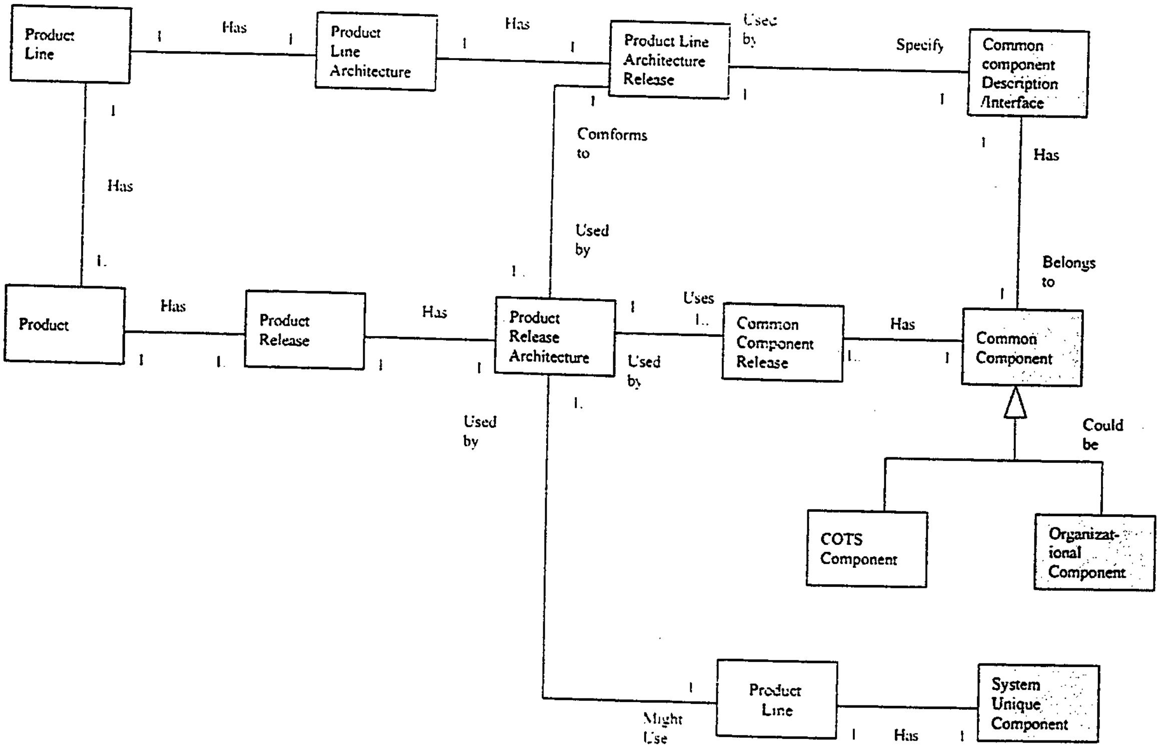


Figure 1 Product Line Views Model

product line, and application engineering is the process used to produce a single product by adapting the domain-wide assets [1].

3 PRODUCT LINE VIEWS MODEL

A product line model that shows different views of a product line, its derived products, and common assets used is presented in this section. It defines entities and relationships between these entities to present product lines. It presents different ways to viewing a product line keeping in mind enhancement, modification, other models, other entities and relationships. Figure 1 depicts the model. The following sections describe the product line views-point model.

3.1. Product Line Overview

A product line is defined as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission [1, 4]. A product line has a group of products associated with it; it has a 1:M relation with its products. A product line has a common architecture associated with it; it has a 1:1 relation with its architecture.

3.2. Product Line Architecture

Product line architecture provides the structural elements and their interfaces by which the system is composed out of the product line [18]. Products are customized using the product line architecture. Product line architecture might evolve during the product line life cycle. New releases of the product line architecture could be seen and this is due to change in customers' requirements, new technologies, design fixes, etc. It has a 1:M relationship with its releases. The early releases of product line architecture specify the common components used in the product line architecture; they could specify the functionality needed by these components and might specify their interfaces. An M: N relationship is established between product line architecture release and common component description/interface. After common components are developed, later releases of product line architecture might refer directly to common component releases. A product line architecture release is used by many products' releases; it has a 1:M relationship with their architectures.

3. 3. Products

Products in a product line are engineered through customization from base requirements, standard product line architectures and integration of common components, and might use system unique components. Each product is associated with its releases. Each product release has architecture associated with it called product release architecture. Product has a 1:M relationship with its releases, whereas, product release has a 1:1 relation with its architecture.

3. 4. Product Release Architecture

Product release architecture is derived from the product line architecture release and must conform to the product line architecture release. It uses many common components described by the product line architecture release; for each common component used, it uses one of the releases of that component. In addition, it might use many system-unique components; for each system-unique component used, it uses a release of that component.

3.5. Components

Components are the building blocks of products in a product line and are classified into two categories: a common component and system-unique component. A common component is used across products of a product line and could be a commercial-off-the-shelf (COTS) component or an organizational component. Organizational components refer to common components developed by the product line organization. They could be developed internally by the organization owning the product line or externally by a different organization within the business unit of which the organization is a part. A system-unique component is used in specific products. Both types of components, common and system-unique, could have releases associated with them and have a 1:M relationship with their releases. They are used in many product releases and have an M: N relationship with product architecture release.

3. 6. Viewpoint Attributes

Entities in the viewpoints have some interesting attributes. Table 1 represents the viewpoint and attributes template. Organizations that adopt the product line approach might be interested in other attributes; these attributes can be added to the table. The attributes listed in table 1 are used to support the views described at section 4 in this paper.

4 PRODUCT LINE VIEWPOINT MODEL

In the product line approach, product Lines share several different views that are interesting to management, system developers, and a reuse team. Other interesting views might be possible.

4.1. Management View

Management of an organization that adopts the product line approach has authority, vision, and leadership. It manages the development of products in a product line. They manage staffing, training, cost, directions, and schedules through the product line cycle. They have a clear vision about the direction of a product line. They interact with customers and make business decisions.

Management in the product line approach can be interested in the products derived from a product line, customers of these products, and customer contact persons. Also they can be interested in cost, contact persons, time intervals, and staffing for products and assets used in these products.

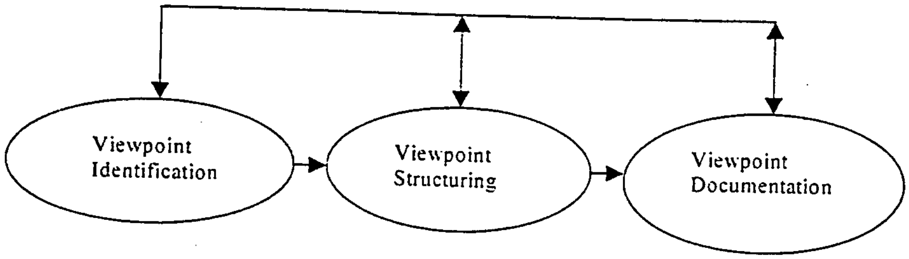


Figure 2. Viewpoint Development Phases

Table 2. Experimental Model Phases for Product Line Validation

Phase	Function	Data Collection Methods
1. Adoption	Assessment	Survey, Legacy
2. Planning & Management	Measurment & Control	Survey, Legacy
3. Utilization	Monitring	Case Study, Project Monitor
4. Expansion	Adaptation	Case Study, Survey, Legacy

This data is supported by the mode. Management can use this data when producing a new product of a product line, negotiating with customers, and assessing the benefits of adopting the product line approach.

The structural of management view and its relationships presented by the model answers questions related to what are the products of a product line and assets used in these products. Attributes used in model's entities answer questions related to who is the customer, contact person, time interval, cost, staffing, etc., of products in a product line.

4.2. Reuse Team View

A reuse team of an organization that adopts a product line approach supports reuse across product lines. They support reuse of components through asset identification. With systems developers they ensure successful use of asset bases in and across product lines. They assess the reuse level across product lines. Reuse team can be interested in viewing product lines, their derived products, and reusable assets (product line architectures and components) used in a product line. They can also be interested in the number of times an asset is reused, and the type of components used in a product line.

The structural of reuse team view and its relations presented by the model shows products of a product line and assets used in these products. Attributes used in the model's entities answer question related to the type of components used, number of times an asset is reused.

The reuse team can use this information through asset identification, ensuring a successful use of asset base in and across product lines, and assessing the level of reuse.

4.3. Systems Developers View

System developers in the product line approach are also interested in viewing product lines, their derived products, the product line architecture, its evolution, assets used and their evolution, the operating system(s) and platform(s) are used, components types, their interfaces.

The structural of system developers view and its relations presented by the model shows the products derived in a product line, the product line architecture, its evolution, components used and their evolution. Attributes used in the model's entities answer questions related the contact person of an asset, components interface, component type, operating system(s) and platform(s).

4.4. Viewpoints Development

We used the method called VORD [17] for the development of viewpoints. Also, this method is principally intended for requirements discovery and analysis, it includes steps that help to translate this analysis into a viewpoint. We considered only the first three stages of the VORD method concerned with viewpoint identification, structuring, and documentation.

a- *Viewpoint Identification* involves discovering stakeholder viewpoint and identifying the specific attributes, tasks, and sub-viewpoints.

b- *Viewpoint Structuring* involves grouping related viewpoints into a hierarchy. Common viewpoints are provided at higher levels in the hierarchy and are inherited by lower-level viewpoints.

c- *Viewpoints documentation* involves refining the description of the identified viewpoints.

Viewpoints and attributes information in VORD are collected using standard forms. The form used for viewpoint information (the viewpoint template) and attributes information (attributes template) are shown in Table 1.

The viewpoints and attributes templates, as well as the viewpoint hierarchy diagrams are developed during the three phases shown in Table 1. The templates are used to structure the information collected. and in general a template cannot be completely filled in during single activity.

5 EMPIRICAL MODEL FOR PRODUCT LINE VALIDATION

In this section an experimental integrated model for product line pilot project planning, measurement. and assessment is presented. This section discusses how qualitative and quantitative process and product line goals are established based on customer and business needs. The process of flow-down of goals to the level of processes and the experimental pilot model is described. Table 2. presents the empirical and engineering model phases for product line validation.

5.1. Making the Product Line Adoption Decision

Product line adoption is defined in the context of an organization rationale to agree, sponsor, commit, or allocate resources for initiating a product line plan or project. Product line utilization is defined in the context of an organization as the creation of assets with the specific "intention" to be reused as well as the utilization of assets that had been specifically created with the "intention" of being reused. Product line management is defined in the context of an organization that manages the creation, utilization, and evolution (i.e., maintenance) of reusable assets. The application of software reuse technologies to planned products (both new and existing) and planned product lines is an indicator that software reuse adoption is strongly correlated with organizational opportunities.

Most software development organizations operate according to marketing and finance strategies. An organization wishing to improve its financial status may look for new or extended opportunities in software product markets. Product line is one possible approach that may be used to leverage decreased time to such markets with decreased effort and increased product quality.

So the first step is to make the product line adoption decision based on some empirically validated software reuse reference model (RRM) [Nada 97]. This in turn will lead to a set of decisions balancing market opportunities with market risks. This step will also identify reuse opportunities, reuse objectives, costs, constraints, and options.

For adoption decision organizations conduct an analytical study to decide either to adopt certain product line process or technology or not. This study collects both qualitative and quantitative benchmark data on the product line approach.

The adoption phase includes several steps to evaluate the technical and organizational aspects of the introduced product line process or technology.

5.1.1 Organization context

Organization context describes the environment in which the organization exists or existed when it launched the product line effort. The following lists common factors that are used in the adoption phase to evaluate the existing environment before applying the product line approach. The following factors will be used to record and evaluate the context environment of organizations adopted the product line approach. Also it used by organizations exploring the transition to the product line approach.

Process or technology objective. To adopt the product line approach; the objective of developing product lines needs to be addressed and defined. This includes defining the scope of the product line, how long the organization has been building product lines, and the product line life cycle.

Costs/benefits. Organizations that already adopted these processes or technologies should have data related to the costs and benefits of this adopting. Organizations that are thinking to adopt a software reuse approach might not have data about the cost of adopting this technology, but the benefits of software reuse approach should be defined. Cost varies based on the size and the number of products in the organization, the technical experience, organization structure needed, skills and training, and tools.

Commonalties and variabilities. Organizations exploring the transition to software reuse approach should identify which products can be considered and what their commonalties and variabilities

Common architecture. Organizations exploring software reuse approach should consider the feasibility of common architecture for their products. Also the style of the architecture might be defined, e.g. layered architecture, client server architecture, etc.

Assets used. In software reuse development approach products are assembled using common set of assets and might use system unique assets. Assets could be domain models, communication protocol descriptions, user interface descriptions, code components, type of common components that developed internally or by using Off-The-Shelf "COTS" components, application generators, domain knowledge, test plans and procedures, requirement descriptions, performance models, metrics, etc. Organizations adopted the software reuse approach records the common assets used in their products. Organizations exploring the transition to the product line approach should define what are the common assets exist.

Level of reuse. One of the benefits of adopting software reuse is increasing the level of software assets reuse in organizations. Organizations adopting reuse approach should have or find other organizations data related to the percentage of reuse achieved in adopting the this approach. Also the type of reuse used, for example, horizontal reuse or vertical reuse. Horizontal reuse represents wide domain width reuse, i.e. a component that can be used in many applications. Vertical reuse represents a narrow domain width reuse, i.e. a component that can be used in one application.

Organization structure. The organization's structure for developing one-at-a-time systems might not be suitable to product line development. Adopting a product line approach has an impact on organization structure. This factor defines the impact of the new structure needed to adopt the product line approach. The impact might be low, medium, or high.

Process. Process used in developing one-at-a-time systems will not be suitable to the product line development. As part of adopting reuse technology, existing process might be modified and new processes need to be in place, e.g. customer interface process, software development processes, etc. This factor defines the impact on the organization processes by adopting new approach, what type of the processes need to be changed, and what type of new processes needed.

Training. Transitioning to new processes or technology requires skilled personnel to achieve a successful transitioning. This factor defines the type of training needed, e.g. in house training, external consultant, etc. Also it defines who needs training, e.g. management, systems developers, etc.

Tools. This factor defines which tools are needed in software development, e.g. tools to assemble products, configuration management tools, tools to record the progress of the product line development, etc.

Software reuse assessment is the main function of this phase. Historical methods are used to collect data, e.g., survey and/or legacy

5.2. Product Line Planning

Organizations use this phase as a plan for the transition to product line software development approach. Organizations can use this phase to record, evaluate, and assess the planning for the product line approach. Organizations intending to adopt software reuse use this phase to put the software reuse in practice.

The following include the implementation plan for software reuse approach; a list of common factors is described in this section as part of the planning phase.

Management Support. Building software products is not just an engineering agenda, it precipitates changes in personnel, personnel management, incentives, customer interface, scheduling, budgeting, and a whole host of management practices. It is a new vigorously and actively supports the transition, the effort will fail. Software reuse strategy means that organizations and managers have less direct control over their product developments and increased dependency on other organizations to understand their requirements and provide acceptable solutions. Giving up this control and the necessary dollars to support product line technology and application development may be difficult. Organizations adopted the software reuse approach should record their experience of the management support, evaluate, and assess that support.

Cultural change. The software reuse concepts should be defined and understood by people of organizations adopting this new approach. A particular attitude that had to be overcome was the one-at-a-time mentality of building a system for its own sake rather than as a contributing effort to the organization's strategic goal of fielding and building up a base set of core assets. Software reuse terminology should be defined and understood across organization.

Organization structure. Adopting new technology or process has an impact on the organizational structure. For example organizations develop product line has a structure different than organizations develop one-at-a-time systems. Some organizations has a product line structure where a marketers group relate product line capabilities to prospective customers; relate customer needs to asset and application developers. A core assets group develops architecture and other assets for product line. An application group deliver systems to customer. There are different players in the product line approach and they should have different skills to launch the product line approach. Transitioning to the product line approach requires the organization's structure and players in the product line approach to be defined.

Training and processes. Transitioning to software reuse involve education and training on the part of management and technicians. Managers need to support the business motivation and strategy of the software reuse approach. They need to understand and role of the infrastructure technologies, understand how to monitor progress and identify potential problems within their area of the program. Different type of training might be needed; Formal training, on-the-job mentoring from external consultants, etc.

New processes are needed to develop a product line is different from processes used in developing one-at-a-time systems. These processes might be customer interface processes, development process, resource ownership processes, etc.

Training and processes changes should be defined in the transition to the product line approach.

New technologies. Technologies allow organizations to stay a competitive edge. Some of the technologies, for example, used in the production of product line are domain engineering and application engineering. Domain engineering used to create artifacts useful across the entire product line. Application engineering is used to produce a single product by adopting the domain-wide assets. Other technologies, for example, using CORBA, COM, etc. These technologies need to be defined in the transitioning to software reuse development approach.

Tools support: Using tools to support the new development approach increase organizations' productivity. Some organizations use tools that are used to assemble products together. Others use tool to capture domain knowledge, etc. These type of tools used needs to be defined in the transition phase.

Software reuse measurement is the main function of this phase. Historical methods are used to collect data, e.g., survey and/or legacy.

5.3. Utilization and Management

Product line utilization is defined in the context of an organization as the creation of assets with the specific "intention" to be reused and the utilization of assets that had been specifically created with the "intention" of being reused.

The next step is to decide upon the levels of the RRM utilization and management and to look closely at any significant changes or impacts on both top and middle management. This step includes the assessment of an organization's willingness to adopt the RRM, the implementation levels, and the incremental investment strategies.

5.3.1 The Product Line Utilization.

Asset Utilization The objective of processes in this family is to utilize existing assets in software development and evolution (i.e., maintenance) activities. The processes for this family consist of developing or selecting criteria for asset identification, modifying or tailoring selected asset(s), and integrating the selected asset in the system under development or evolution

This step is the actual production phase by applying evolutionary approach (Boehm Spiral Life-Cycle Model) to the reuse plan implementation. Our early research results have shown that software development organizations at a high success (capability) level usually carry out several pilot (experimental) projects to help them in the construction of a prototype repository, component model definition, components classification scheme definition, domain model, common architecture, and product-line as follows:

I. Develop a prototype (pilot project)

II. Learn and evaluate of risk versus opportunities

(including assessment of effort, quality, schedule, tools, and procedures)

III. Expand prototype to a safer version of product line with the necessary adjustment

Repeat step (II) and (II) until you achieve a stable product line version.

This approach to the successful learning and evolving the RRM within an organization is like the Boehm Spiral Life-Cycle Model [8] applied to the RRM implementation plan.

5.3.2 Product Line Management

Reuse management is defined in the context of an organization that manages the creation, utilization, and evolution (i.e., maintenance) of reusable assets.

Asset Management and Control: The objective of processes in this family is to develop and organize collection(s) of quality reusable assets, define and develop services and capabilities to access these assets (i.e., for asset utilization processes), and establish, support, and enact a broker role for asset developers (i.e., from asset creation) and asset consumers (i.e., from asset utilization).

The reuse management and control is based on the classic plan, enact, and learn cycle. The plan, enact, learn cycle in the reuse management idiom is based on the following principles as described in the STARS CFRP [11].

Software reuse monitoring is the main function of this phase. Observational and historical methods are used to collect data, e.g., survey, case study, historical analyze and/or legacy

5.4. Product Line Expansion

In this phase, organizations look for new product opportunities and asses the customer needs and reuse future plan.

Determining and evolving the future objectives, strategy, and scope of a reuse program, resulting in selection of a set of suitable domains and products lines in which to apply reuse within an organization. Planning, establishing, monitoring and evaluating Reuse engineering idiom (asset creation, asset management, and asset utilization) projects addressing the selected domains and product lines. Looking for new market opportunities, market analyze, and assess the future financial plans.

Software reuse adaptation is the main function of this phase. Observational and historical methods are used to collect data, e.g., survey, case study, historical analysis and/or legacy.

6 REPOSITORY SUPPORT

Organizations adopting the product line approach can use a repository to implement the model. The repository supporting the product line approach can capture the entities and their related attributes, and the relationships between these entities to covey the model's views. A web-based repository is a good choice to implement the model. It provides and easy access for many users internally or externally to organizations developing product lines. The Web-based repository can model the entities, some of their related attributes, and relationships as Hyper-text links to present a complete picture of the entire product line.

7 CONCLUSIONS

Organizations that produce similar systems are moving towards implementing the product line approach. Products in the product line approach are engineered through customization from base requirements and product line architectures, integration of common components and system-unique components.

The model described in this paper is intended to capture a view of the product line, its derived products, and assets used in the product line. The model is defined to present views interested to management, system developers, and a reuse team in the product line approach.

REFERENCES

1. Bass, L., Clements, P., Cohen, S., Northrop, L., and Withey, J., "Product Line Practice Workshop Report", June 1997, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html.
2. Cohen, S., Fridman, S., Martin, L., Poyer, T., Solderitsch, N., and Webster, R., "Concept of Operations for the ESC Product Line Approach", Sept. 1996.
3. Brown, A., and Wallnau, K., "Engineering of Component-Based Systems", Proceedings of the 2nd IEEE International Conference on Engineering of Complex Systems, 1996. IEEE Computer Society Press 1996.
4. Brownswod, L., and Clements, P., "A Case Study in Successful Product Line Development", Oct. 1996, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
5. Clements, P., "Report of the Reuse and Product Lines Working Group of WISR8", Aug. 1997, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
6. Fraks, W., "Success Factors of Systematic Reuse", IEEE software, Sept. 1994.
7. N. Nada, Software Reuse-Oriented Functional Framework, Ph.D. Dissertation, George Mason University, fall 1997.
8. Perry, D., "generic Architecture Descriptions for Product Lines", http://www.bell-labs.com/usr/dep
9. D. Rine and R. Sonnemann, "Investments in Reusable Software: A Study of Software Reuse Investment Success Factors", The Journal of Systems and Software, Vol. 41, pp. 17-32, 1998.
10. D. Rine and N. Nada URL-http://www.gmu.edu/depts/survey.
11. Shaw, M., and Garlan, D., "Software Architecture", Prentice-Hall, Inc., 1996.
12. Software Technology for Adaptable, Reliable Systems (STARS), "STARS Conceptual Framework for Reuse Process (CFRP)", CDRL A018, Oct. 1993
13. Sommerville, I., Software Engineering, 5th Edition, Addison-Wesley, New York, (1996).
14. The Software Evolution and Reuse Consortium, "Solutions for Software Evolution and Reuse", SER Deliverable SER-D2-A, 1995.
15. Withey, J., "Investment Analysis of Software Assets for Product Lines", Nov. 1996, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
16. M. Zelkowitz, "Experimental Models for Validating Technology", IEEE Computer, May 1998.
17. Kotonya and Summerville, Requirements Engineering, Wiley, 1992
18. G. Bootch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1999.

A Knowledge-Based System for Software Reuse Technology Practices

N. Nada and L. Luqi
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943
Phone: (831) 656-4075
Fax: (831) 656-3225
Email: nnada@cs.nps.navy.mil

D. Rine, E. Damiani, S. Tuwaim
George Mason University
Computer Science Department
Fairfax, VA. 22030
Phone: (703) 993-1530
Fax: (703) 993-1710
Email: drine@cs.gmu.edu

ABSTRACT

The practicing and researching software engineering communities are still in need of professional practice resources and on-line tutoring systems that can be easily used to identify lessons learned and reuse experiences from successful enterprises based upon a validated software reuse reference model for the software reuse process within the general software development life-cycle. This paper presents a public Case-Based System using a validated Software Reuse Reference Model (CBS-RRM). A CBS-RRM allows the software engineers to improve reuse practices by being tutored with selected course material based on the user profile. This material is combined with actual practice-based knowledge derived from different positive cases from software development organizations' reuse practices. A CBS-RRM provides software engineers with a way to be tutored using positive lessons learned by other organizations. Our research focuses on achieving more effective means for software development organizations to find alternative educational (training) solutions to problems in successful practice of reuse. The paper focused only on the CBS module.

Keywords

Case-Based Reasoning Systems, Intelligent Tutoring, Distance Learning, Learning Environments, Web-Based Training Systems.

1 OVERVIEW

1.1 Intelligent Tutoring Systems

Traditional intelligent tutoring systems are based on the assumptions that a student's thinking process can be modeled, traced, and corrected.

Based on the principles of Computer Assisted Instruction (CAI), intelligent tutoring systems would allow for a generic model that can be used for any individual. There are four main components of an intelligent tutoring system. The student module (1) consists of the incorrect and incomplete knowledge that a student begins with. The expert module (2) contains the correct, expert-like knowledge that is to be transferred and learned. This transfer of learning occurs as a two-way communication process, made possible through (3) the graphical user interface (GUI). The pedagogical element (4) is the basis of the instruction, and it determines what instruction will be given at which point. Some intelligent tutoring systems go further, and incorporate full simulation as part of the instruction.

The term "intelligent" refers to the system's ability to know what to teach, when to teach it, and how to teach it. It must have the capacity to understand, learn, reason, and problem solve. It must be capable of identifying a student's strengths and weaknesses and establish a training plan that is consistent with these results. It can pick up relevant learning information from the student (such as learning style), and apply the best means of instruction for that particular individual. Throughout the instruction, the system makes judgments about what the student knows and how well she/he is processing the information. The instruction can then be tailored to the student's needs. [31, 3, 6, 22]

1.2 Software Reuse Reference Model (SRRM)

In recent years, reusability has become an important factor in the process of software development. In fact, the availability of reusable assets in development phases provides valuable support to design and implementation with software architectures by improving productivity, quality, and time-to-market [14]. Industry has demonstrated that reuse of software assets will provide a basis for dramatic improvements in quality and reliability, speed of delivery, and in long-term decreases in costs for software development and maintenance. Some researchers estimate that even with a less than 50% reuse rate, component-based software development leads to reliability improvement as much as ten times that of development that is not component-based [7].

Opportunistic software asset reuse will not always succeed if it is not based upon a supporting reference model for developing software [33]. Hence, a Software Reuse Reference Model (SRRM) may be considered as a key starting element to implement, realize, and quantify such savings. The SRRM needs to include both technical and organizational activities required to implement reuse successfully.

1.3 Case-Based Systems (CBS)

Case-Based Systems (CBSs) offer a knowledge architecture system for managing, sharing and accessing knowledge. A CBS unifies many previous forms of knowledge management into a single intuitive mechanism. CBSs support such diverse knowledge types as structured data, free-text documents, activity patterns, and expert system knowledge bases. CBSs unify access methods such as query-by-example, free-text retrieval, decision trees, and case-based reasoning (CBR).

There are two primary benefits to the use of CBSs. The first is to provide access to a broad spectrum of on-line knowledge through a single access method. The second is that CBSs are fundamentally superior for certain types of access, especially ad-hoc searches for relevant knowledge to help answer a question or resolve a problem.

The CBS approach uses the technique of comparing a current situation (e.g. company profile) to a library of known solutions (e.g. successful professional practices). CBS has been applied to a range of classification and construction tasks. It is particularly useful in tasks where a formal set of rules, patterns, or algorithms for generating solutions is difficult to obtain, but where examples of correct solutions are readily available. These "previous solutions" are stored as "cases" in a case base. The case base can be used for multiple purposes, including training and human and automated decision-making. Because of this, a CBS can keep pace with a changing environment by adding and improving cases, eliminating the need for repeated software upgrades performed by knowledge engineers. Because of the simple knowledge representation, using case study templates and patterns, little expertise is required to maintain the CBS. The CB manager does not need to be a programmer [1,5, 24, 30]

1.4 CBS and SRRM Correlation

It is necessary for software developers to have systematic procedures supported by a CBS and a validated SRRM to provide a real starting point for good software assets reuse and adoption decisions, utilization decisions, and management activities. In addition to a SRRM, an organization interested in moving into a reuse-oriented software development methodology also needs more detailed knowledge about how to implement the SRRM in the organization. Hence, access to a CBS with this more detailed knowledge would be very useful.

It is important for software reuse practitioners and new enterprises that are interested in adopting software reuse to access lessons learned, access more detailed knowledge about how to implement the SRRM in the organization, and access reuse experience of successful enterprises based upon a validated SRRM for the software reuse process. Accessing these three kinds of knowledge is but a first step in an iterative software improvement environment. Usually, it is important to know what lessons and experiences lead to improved software development. But it is equally important to be able to implement and practice the skills behind these lessons and experiences so that, by doing and not just knowing, measured improvements will occur. Hence, a second step is building an educational environment, based upon individual tutoring, where the knowledge accessed in the CBS can be incorporated into individualized learning based on implementation and practice of those skills that will, in turn, lead to measured improvement. Measured improvement can, in turn, lead to increased software assets quality and increased process productivity. Section 2.3 describes such a total CBS-SRRM educational environment where learning based upon individualized tutoring can take place.

The existence of a publicly accessed Reuse CB (National Reuse CB), via our CBS-RRM will help software industry and academia capture best practice-based knowledge derived from different software development organizations' reuse programs and activities. This reusable set of best practices available by use of our proposed CBS-SRRM could provide software industry and academia with a systematic way to capture and access the lessons learned by other organizations. This will promote recurrence of good reuse practices and improve current reuse processes by increased software quality and decreased effort and time to market.

Having a set of case studies that can be used to derive solutions to reuse problems from prior lessons learned will help to carry out the following: (1) Describe current problems and identify ways to avoid them in the future. (2) Predict opportunities and possible successes in applying reuse. (3) Derive new knowledge from ongoing research projects. (4) Better leverage best reuse practices. (5) Avoid unnecessary risks. (6) Better justify technical and business reuse decisions.

Our assertion is that the case studies and lessons learned would be reused more often if organizations that have successfully adopted, utilized, and managed reuse could indirectly help organizations with similar environments, problems, or situations, and are interested in adopting or researching software assets reuse, locate the information about best software assets reuse practices and decisions about whether or not to adopt, utilize and/or manage software development based upon reuse.

2 WHAT IS MISSING

Referring to our previous research in the area of software assets reuse [Nada 97, 27], the practicing and researching software engineering communities are still in need of the following professional practice resources:

- A publicly accessed CBS for the software engineering community that can be easily used to identify lessons learned and reuse experiences from successful enterprises based upon a validated software reuse reference model for the software reuse sub-process within the general software development processes.
- Use of an applicable, conceptualized, effective, and validated software assets Reuse Reference Model that considers and incorporates all technical and non-technical aspects of the software reuse process.
- On-line Software Reuse Self-Assessment system.
- On-line Software Reuse Individualized Distance Learning system.
- Identification of effective software assets reuses processes and products metrics.
- Identification of standardized reuse practices, i.e. systematic software reuse methodology.

3 CBS-SRRM KNOWLEDGE BASED TUTORING SYSTEM

Based on the principles of Computer Assisted Instruction (CAI), CBS-SRRM tutoring systems would allow for a generic model that can be used for any individual who is involved in software development and engineering [31].

3.1 CBS-SRRM Overview

Our current project, funded by the NSF, investigates effective public Case-Based System (CBS) tool-kits using a validated Software Reuse Reference Model (SRRM).

CBS-SRRM allows the software engineer to improve reuse practice by capitalizing on effective practice-based knowledge derived from different software development organizations' reuse practices. CBS-SRRM provides software engineers with a way to utilize lessons learned by other organizations. The system also promotes recurrence of good reuse practices.

The research focuses on a more effective means for software development organizations to find alternative solutions to problems in successful practice of reuse. We demonstrate that developing a CBS-SRRM that will allow software developers to learn how organizations similar to theirs have successfully adopted, utilized and managed this technology can support improved reuse practices. The plan is to research, develop, and make publicly available what our affiliates and we have learned through our evolving set of case studies, surveys. interviews, and experimental results. This plan is carried out by researching and developing a publicly accessible reuse practices CBS for the software engineering community, using lessons learned and reuse experiences from successful enterprises based upon a validated SRRM that incorporates important technical, organizational, and cultural factors needed in adopting, utilizing, and managing reuse technology.

3.2 CBS-SRRM Objectives

The main objective of this research is to develop a tutoring system including a knowledge-based web-based distance assessment module that is technically supported by Case-Based Reasoning (CBR) technology.

The objective is to motivate software developers to access a web-based tutoring system including an assessment module that will help them improve their software development process using reuse practices. The practical implication is to provide trainees with a demonstration of a more efficient, more effective, and publicly accessed assessment and teaching package that will enhance their learning outcomes, increase their productivity, and improve their products' quality in shorter time.

We have collected, and continue to collect, data from industry on actual processes used and experiences with software reuse. This data is collected and then presented on the Web in a standard form based on a validated model. The CBS-RRM also provides interface to allow users to describe their own environment and objectives and to receive the data corresponding to the recorded projects that best match their profile. Work such as this can be of great value for developers who are under increasing economic pressure to avoid building each new system from the ground up. It is also of value to the research community as an empirical basis for the validation of claims and methods related to software reuse.

3.3 A CBS-SRRM Tutoring System

There are four main components of the tutoring system. (1) The student profiling module that will qualify the student for a certain software engineering domain, and identify the student's or trainee's (user's) organization size. (2) The assessment module that will examine and assess the user's previous software reuse experience and his/her organization reuse potential, capability, RRM level, and the depth of users's knowledge and experience in reuse. This step will be followed by a pre-test to evaluate the student/trainee background knowledge on reuse; our prototype can identify 3 levels: initial, middle, or advanced. Based on the outcome of the previous two modules and the results of the user's pre-test, the student will be assigned to a certain level of training material. (3) The CBS module will use the profiling information to match the student with several case studies, and present the best software reuse practices that have been used by similar organizations. This module contains the correct, expert-like knowledge that is to be transferred and learned. (4) The fourth module contains the course material that fulfills the users' needs and matches their profile.

The current CBS-SRRM tutoring system allows software developers to learn how organizations similar to theirs have successfully adopted, utilized, and managed improved reuse practices enterprises based upon a validated SRRM that incorporates important technical, organizational, and cultural factors needed in adopting, utilizing, and managing reuse technology. We are researching, developing, and making publicly available what our affiliates and we have learned through our evolving set of case studies, surveys, and interviews, thereby making it available to the whole software engineering community.

3.4 CBS-SRRM Architecture

Using a web-based Distance Assessment and Tutoring system combined with the CBR system will provide tools to allow students and supervisors to have a good educational system to improve the individual's skills and knowledge in software reuse. The CBS-SRRM Architecture is depicted in Fig. 1. The remaining part of this paper will focus only on the CBS module.

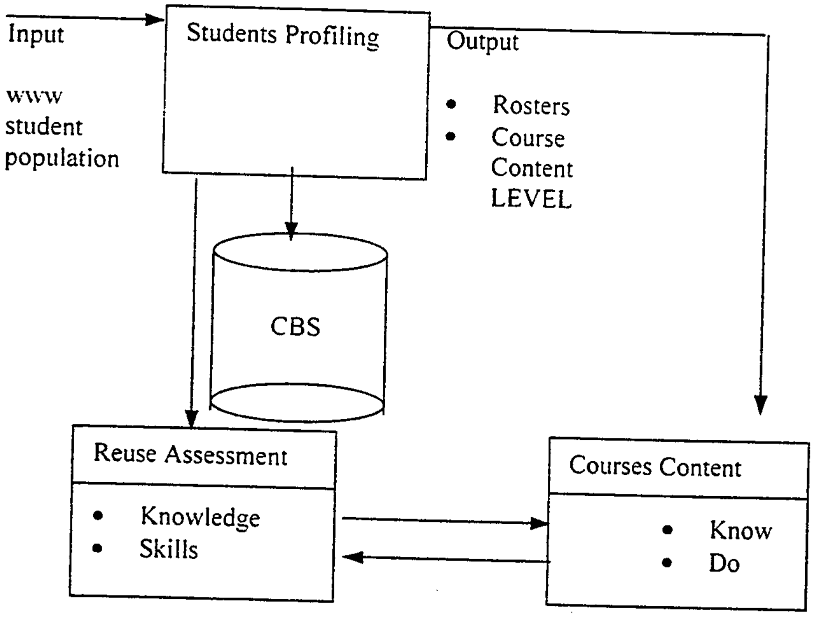


Fig. 1 CBS-SRRM Architecture

3.4.1 Searching Requirements of the Best Practices CBS

Believing that analogues may provide a way to predict results based upon what has been true in the past, the CBS's searching mechanism will be developed along the lines of searching systems. It will maintain a CB of cases that represent the performance of best-practiced software reuse. When the partially known profile of a new organization is presented to the CBS, the search engine will search the CB, find the case(s) of organization(s) and its/their profile(s) that is/are most similar with the profile of the new organization, and finally predict the level of practice of the organization in the CB that will be the level of practice assigned to the new organization. We adopted the following CBS Architecture (Fig. 2) [37].

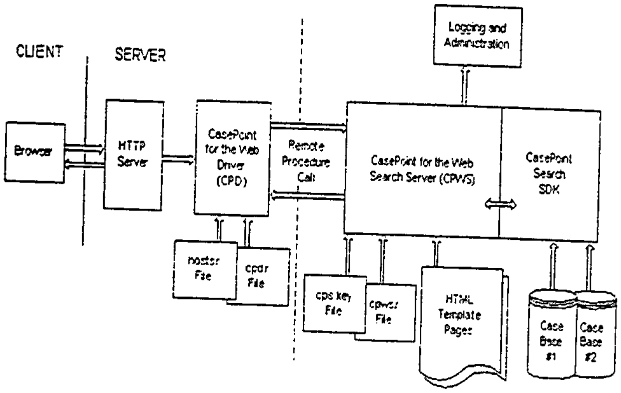


Fig. 2 CBS Architecture

3.4.2 Reuse Practice Cases: Development of CB Study Subjects

The participant subjects are software development organizations who (1) have already been case study participants and who are initially in our CB of best practices, and (2) are considering adopting, utilizing and managing software reuse. Nada worked on the identification and evaluation of new CB subjects. Initially, each organization will constitute a case that contains the profile of certain user attributes. Cases that include all of this information will comprise the space of CB cases. Cases that are lacking the final software reuse practice level assigned, but contain at least a subset of the remaining information, will be considered as test (input) cases. The choice of organizations that will comprise the CB cases and the organizations that will comprise the test cases will be 'pseudo-random'.

The CBS's task, researched and developed by our team, will be to find an appropriate value for the level of reuse practice attribute of an input case; therefore, this attribute is considered the solution data for a particular case in this domain.

3.4.3 Matching Requirements of the Best Practices CBS

During testing of the CBS's predictive power using new subjects, the CBS search engine will need to use matching methods [2,38, 9, 23,34]. Based on the methods used to establish the similarity between certain new test cases and current CB cases, the CBS will compare corresponding features one at a time.

Each test case will contain six features. The first two of these features will be used to identify the particular organization type to which a certain organization belongs. The remaining four features will denote the partially known organization type software reuse practice level of the same organization, and they will be used as indexing features. These four features are the organization's reuse practice levels in the first, second, and third stages of reuse adoption, utilization, and management, and the organization's practice level at the end of the evaluation period. Given this partially known organization type's reuse practice level, i.e., given a test case, the CBS's task will be to predict the organization's practice level within the class of the given organization type.

This will be done by using the case CB in order to find the case or cases that are most similar to the test case. Similarity will be determined by comparison of corresponding indexing features. For example, corresponding indexing features with identical numerical values will receive a similarity count of 1 while corresponding features such that the absolute value of their difference is greater than, e.g., 10 percent will receive a similarity count of 0. If the difference is less than, e.g. 10 percent then the similarity count will be a numerical value between 0 and 1. The sum of the similarity counts for each feature will constitute the degree of similarity between two cases; therefore, the maximum possible match value between two cases will be equal to the number of case features. For example, the previously shown CB and test cases exhibit a certain (e.g. 70) percent matching confidence since their degree of similarity is 70 percent.

4 CONCLUSION

This paper focused only on the CBS module. The paper presents a public CBS using a validated Software Reuse Reference Model (SRRM). A CBS-SRRM allows the software engineer to improve reuse practice by being tutored with selected course material based on the student profile. This material is combined with actual practice-based knowledge derived from different positive cases from software development organizations' reuse practices. A CBS-SRRM provides software engineers with a way to be tutored using positive lessons learned by other organizations. Our research focuses on achieving more effective means for software development organizations to find alternative educational (training) solutions to problems in successful practice of reuse.

Our future work will focus on presenting and integrating a comprehensive CBS knowledge-based tutoring system that supports distance learning and reuse self-assessment in combination with CBR and empirically validated SRRM.

REFERENCES

1. Aamodt, A., and Plazas, E. "Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches." AI Communications 7(1) (1994): 39-52.

2. Goguen, J.; Nguyen, J.; Meseguer, J.; Luqi, L; Zhang. D.; Berzins, V. "Software Component Search," Journal of Systems Integration (special issue on Computer Aided Prototyping) Vol. 6. No. 2 (1996): 93-134.

3. Hall, P. & Wood, P. "Intelligent Tutoring Systems: A Review for Beginners." Canadian Journal of Educational Communication 19(2) 1990: 107-123.

4. Jaber., K., Nada, N., and Rine, D. "Towards the Design and Integration of Multi-Use Components." Proceedings of the International Conference on Information Systems Analysis and Synthesis, July 1998.

5. Kolodner, Janet L. Case-Based Reasoning. San Francisco, California: Morgan Kaufmann, 1993.

6. Laurillard, D. "The Pedagogical Limitations of Generative Student Models." Instructional Science, 17 (1989): 235-250.

7. Lim, W. "Effects of Reuse on Quality, Productivity, and Economics." IEEE Software September 11(5) (1994): 23-30.

8. Lim, W. Managing Software Reuse. Englewood Cliffs, NJ: Prentice-Hall. 1998.

9. Luqi, Y. Lee. "Towards Automated Retrieval of Reusable Software Components." Proceedings of the AAAI Workshop on Artificial Intelligence and Automated Program Understanding, San Jose, CA, July 13, 1992. 85-88.

10. McDowell. A Reusable Component Retrieval System for Prototyping. MS Thesis, Naval Postgraduate School, September 1991.

11. McClure, C. The Three Rs of Software Engineering: Reengineering - Repository - and Reusability. Englewood Cliffs, NJ: Prentice-Hall, 1992.

12. Morisio, M., Ezran, M., and Tully, C. "Introducing Reuse in Companies: A Survey of European Experiences." Proceedings of the 1999 Symposium on Software Reuse, ICSE-99, IEEE and ACM, 1999.

13. Nada, N., Rine, D., and Tuwaim, S. "Best Software Reuse Practices Require Reusable Software Architecture in Product Line Development." Proceedings of the Second Workshop on Software Architectures in Product Line Acquisitions, June 1998.

14. Nada. N., and Rine, D. "Software Reuse Reference Model: Development and Validation." International Conference on Software Reuse, Victoria, Canada, June 1998.

15. Nada. N., Rine, D., and Tuwaim, S. "Practices in Organizational Structure for Software Reuse." Proceedings of International Conference on Information Systems Analysis and Synthesis, July 1998.

16. Nada, N., Jaber, K, and Al-Daijy, E. "A Product-Line Model." Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), Vancouver, Canada, October 1998.

17. Nada, N., and Rine, D. "Modeling and Designing Global Data and Information Systems Under Software Reuse Emerging Technologies." Fourth International Conference On Computer Science and Informatics (CS&I'98), Special Session on Software Reuse, October 1998.

18. Nada, N. and Rine, D. "Component Management Infrastructure: A Component-Based Software Reuse Reference Model." Proceedings of the ICSE98 International Workshop on Component-Based Software Engineering, Japan.

19. Nada, N., Rine, D., and Jaber., K. "Towards Components-Based Software Development." Proceedings of European Reuse Workshop (ERW'98), Spain, November 1998.

20. Nada, N., Rine, D., and Jaber., K. "Using Adapters to Reduce Interaction Complexity in Reusable Component-Based Software Development." Proceedings of the Symposium on Software Reusability (SSR'98), in conjunction with the International Conference on Software Engineering (ICSE'99), Los Angeles, May, 1999.

21. Nada, N. and Jaber, K. "Experimental Model to Validate Software Reuse Technology." Proceedings of the International Symposium on Computer and Information Sciences (ISCIS'99). Izmir, Turkey, October 18-20, 1999.

22. Newman, D. "Is a Student Model Necessary? Apprenticeship as a Model for Intelligent Tutoring Systems." Proceedings of the Fourth International Conference on Artificial Intelligence and Education, 1989. 177-184.

23. Nguyen, D. An Architectural Model for Software Component Search Ph.D. Dissertation. Naval Postgraduate School, December 1995.

24. Riesbeck, C. and Schank, R. Inside Case-Based Reasoning. Hillsdale, N.J.: Lawrence Erlbaum, 1989.

25. Rine, D. and Sonnemann. "Investments in Reusable Software: A Study of Software Investment Success Factors." Journal of Systems and Software vol. 41 (1998): 17-32.

26. Rine, D.. Nada, N.. and Tuwaim. S. "Practices in Organizational Structure for Software Reuse." Proceedings of the IEEE SCI Conference, vol. 1, 1998.

27. Rine, D., and Nada, N. "A Validated Software Reuse Reference Model Supporting Component-Based Management." Proceedings of ICSE98, Japan.

28. Rine, D. and Nada, N. "Software Reuse Reference Model: Development and Validation." Journal of Information and Software Technology, in press, (1999).

29. Rine, D. and Nada, N. "An Empirical Study of a Software Reuse Reference Model." journal submission under review, (1999).

30. Schank, R., Riesbeck, C., and Kass. A. Inside Case-Based Explanation. Hillsdale. N.J.: Lawrence Erlbaum, (1994).

31. Schnackenberg, H. Class Notes, Introduction to Educational Computing (ETEC 560/660), Concordia University, Montreal, Canada, (1999).

32. Sonnemann, R. Exploratory Study of Software Reuse Success Factors. Ph.D. Dissertation, George Mason University, Fairfax, Virginia, Spring, (1995).

33. Sommerville, I. Software Engineering. 5th Edition, New York: Addison-Wesley, 1996.

34. Steigerwald, R., Luqi, L., McDowell, J. "A CASE Tool for Reusable Software Component Storage and Retrieval in Rapid Prototyping." Information and Software Technology England, Vol. 38, No. 9, 698-706, Nov. (1991).

35. Steigerwald, R. Reusable Software Component Retrieval via Normalized Algebraic Specifications. Ph.D. Dissertation, Naval Postgraduate School, December (1991).

36. Herman, J. Improving Syntactic Matching for Multi-Level Filtering. MS Thesis, Naval Postgraduate School, September (1997).

37. Inference Inc., K-Commerce Users Manual.

Integrated Design and Process Technology, IDPT 1999
Printed in the United States of America, June, 2000

RISK ASSESSMENT IN SOFTWARE REQUIREMENT ENGINEERING[1]

Juan C. Nogueira, Luqi, Valdis Berzins
Department of Computer Science
Naval Postgraduate School
Monterey, CA

ABSTRACT

In 1994 Gibbs claimed that "despite 50 years of progress, the software industry remains years—perhaps decades—short of the mature engineering discipline needed to meet the demands of an information-age society." Many researchers have treated the problem using different approaches: tools, formal methods, prototyping, software processes, etc. However, this assertion remains true today. This paper considers the problem from the point of view of requirement engineering and risk assessment. We present an improvement to the evolutionary prototyping process model.

1. Introduction

In complex software systems, reliability is an important aspect of software quality that has been elusive in practice. Since more and more human activities and systems are dependent on software, achieving the appropriate level of reliability in a consistent and economical way is crucial. Software failures inconvenience people at best, and in extreme cases can kill them.

Much reliability research has been conducted studying the behavior of a system after it is operable. This work has strong theoretical statistical foundations and many of these models have been shown to be very accurate. However, post-mortem analysis of the behavior of a system gives insights too late to be useful for software development.

This paper describes a way to improve reliability of systems from the beginning of the process. Studies have shown that early parts of the system development cycle such as requirements and design specifications are especially prone to errors. Problems originating in the early stages often have a lasting influence on the reliability, safety and cost of the system. In early stages we cannot directly assess reliability of products that do not exist yet, but we can assess risks that could contribute in the future to the lack of reliability, quality and usefulness of the system.

Evolutionary prototyping offers an iterative approach to requirement engineering to alleviate the problems of uncertainty, ambiguity and inconsistency inherent in the process. Moreover, prototyping can improve the capture of change in requirements and assumptions during the development process. This effect is particularly notorious in projects involving multiple stakeholders with different points of view.

Computer Aided Prototyping System (CAPS) [1] is a CASE tool that provides a collection of techniques and languages for computer-aided prototyping, including logical assessment of the consistency and clarity of requirements and specifications. CAPS methods involve the use of real-time constraints and abstract modeling to describe the requirements in a clear, precise, consistent and executable format. Prototypes can be applied to demonstrate system scenarios to the affected parties as a way to: a) collect criticisms and feedback that are sources for new requirements; b) early detection of deviations from users' expectations; c) trace the evolution of the requirements; and d) improve the communication and integration of the users and the development personnel.

2. CAPS (Computer Aided Prototyping System)

Real time systems present special difficulties in terms of requirement engineering. Some requirements are difficult for the user to provide and for the analysts difficult to determine. The best way to discover these hidden requirements is via prototyping. CAPS is a tool specially suited for this task. It has a graphical easy to understand interface that maps to a specification language, which in

[1] This research was supported by the US Army Research Office under grant #38690-MA and grant #40473-MA.

turns generates Ada code. The main components of CAPS are:

(a) The prototype system description language (PSDL). PSDL is based on data flow under real-time constraints. It uses an enhanced data flow diagram that includes non-procedural control and timing constraints.

(b) User interface based on a graphic editor with a palette of objects that include operators, inputs, outputs, data flows and operator loops. A search engine helps the designer to find reusable components.

(c) The software database system provides a repository for reusable PSDL components.

(d) The execution support system consists of a translator, scheduling mechanisms, execution monitors, and a debugger.

The prototyping process consists of prototype construction and modification (evolution) based on evolving requirements and code generation. Both construction and modification are exploratory activities with a common target: to satisfy multiple users with different and often conflicting points of view. Requirement engineering is a consensus driven activity in which mechanisms for conflict resolution and traceability of requirement evolution represent critical success factors.

3. REMAP (Representation and Maintenance of Process Knowledge)

The REMAP model [2] represents the conflict resolution of requirements in a multiple stakeholder environment. It is an improvement of the IBIS model introduced by [3]. Figure 1 shows the conceptual model of REMAP.

Requirements are the main input and output of each demonstration of the prototype. Initially, a small set of requirements is collected. The requirements generate controversy between different stakeholders. The argumentation process is covered by the extension to the IBIS model. The primitives of IBIS are issues, positions and arguments. Issues are questions or concerns. Positions represent the points of view of different stakeholders. Arguments can support or object to positions, and are based on assumptions. Design decisions resolve issues introducing constraints, which define design artifacts.

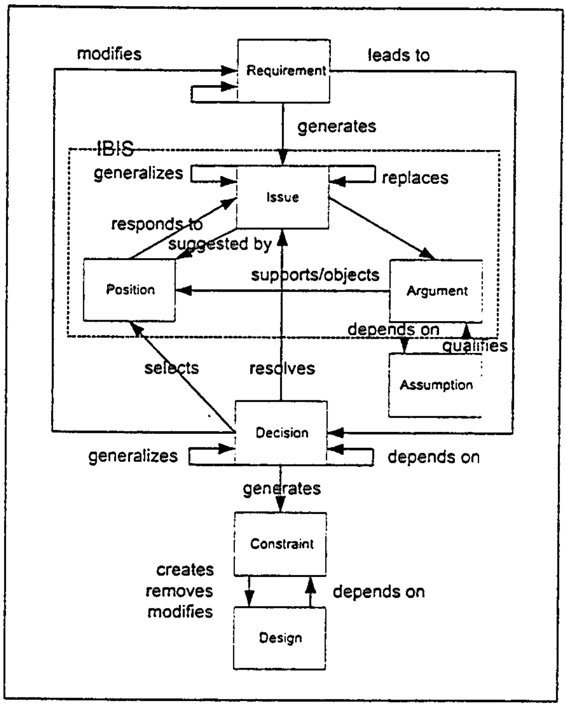


Figure1: REMAP model

The requirement engineering process transforms initial requirements that usually are informal and imprecise into more technical and precise specifications. Specifications are required for practical development purposes and can be understood by engineers. However, they are not well understood by users. So, it is necessary to provide a full spectrum of descriptions. For that reason, the primitives of REMAP have been integrated into the graph model [4] in successive efforts [5] and [6].

4. The Graph Model

The graph model is a data graph model for evolution that records dependencies and supports automatic project planning, scheduling, and configuration management. The evolution process is represented by a graph that at any given moment models the current and the past state of the software system as well as planned future states.

The model views a software evolution process as a partially ordered set of steps. Steps represent activities required to produce the system. A step has states that reflect the dynamic progression of the activity from the moment that it is proposed to the moment it is completed or abandoned.

The graph model has experienced its own evolution process. Luqi [1] introduced a primitive version of the model. Mostov and Luqi [7, 4] refined and elaborated the model. In [4], the notion of hypergraph was introduced to realize automated software evolution in multidimensional phases. Further refinements including scheduling and team coordination, were introduced by [8]. Conflict resolution of requirements and criticisms introduced by Ramesh [2] and Ibrahim [5]. Luqi [9] extended the graph model to a hierarchical hypergraph that improved the traceability of the dependencies and introduced the concept of hyper-requirements. Finally, Harn extended the model to a relational hypergraph model [6].

5. Risk assessment driven software evolution

Experience suggests that building and integrating software by mechanically processable formal models leads to cheaper, and more reliable products sooner. Software development processes such the hypergraph model for software evolution, or the spiral model [10], have improved the state of the art. However, they have a common weakness: risk assessment.

In the software evolution domain risk assessment has not been addressed as part of the model. In the various enhancements and extensions the graph model did not include risk assessment steps, hence risk management remains as a human-dependent activity that requires expertise.

In the evaluation of the spiral model, one of the difficulties mentioned by Boehm was: *"Relying on risk-assessment expertise. The spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk." "...Another concern is that a risk-driven specification will also be people-dependent."* [10].

Many researches have addressed the problem of risk assessment following the perspective of the traditional disciplines. The tools for risk assessment are guidelines for practices, checklists, taxonomies of risk factors and few metrics. All these methods work fine IF carried out by a human educated on risk assessment AND with enough experience. Unfortunately, such resources are really scarce.

From the point of view of software engineering, it is necessary to create a method to support the decision-making process during the early stages of the life cycle, when changes can be made with less impact on the budget and schedule. In our vision, software risk management deals with how to administrate complexity and how to assign resources. We propose to separate risk assessment into three classes: resource risk, process risk and product risk.

Resource risk is the amount of project risk created by threats imposed by available resources. It is affected by organizational, operational, managerial and contractual parameters such as outsourcing, personnel, time and budget. The literature is abundant in this area [11, 12]. Various approaches use subjective techniques such as guidelines and checklists [13], [11], which require the opinion of an expert even when they could be supported by metrics. [12] has introduced a more rigorous method. In this approach, the risk is viewed as a three dimensional entity that depends on schedule risk schedule, cost risk and technical risk.

The process risk is the amount of the project risk caused by management work procedures such as planning, quality assurance, and configuration management. It is also caused by technical work procedures related to the software processes such as requirements, analysis, design, code generation, testing, etc. The more complex a process is, the more difficult it is to manage. More education, training, standards, reviews, and communication are required. Consequently, complexity grows. Software process complexity has been partially addressed by research in terms of subjective assessments about maturity level and expertise [13, 11, 14]. However, we seek a more precise and objective method. Several approaches to study process complexity in a static way have been introduced in the field of management. Simulation can be used to measure the complexity of the dynamics of the processes.

Finally, product risk is related to the final characteristics of the product, its conformance with specifications and requirements, its reliability and customer satisfaction.

We think that there exists a dependency between these classes of risk. The success of the project depends on its own characteristics and in the success of the product and the process. The success of the process depends on itself as well as in the success of the project and the product. And the success of the product depends on itself and on the success of the project and the process. The dependencies among the three areas constitute an equivalence relation because the symmetric, transitive and reflexive properties apply. In our view, this reflects the fact that resources, process and product are different facets of the same entity: the project.

Dealing with threats, the decision-maker can apply the following strategies:

- Risk absorption, which is to assume the consequences of the risk as a constraint.
- Risk avoidance, which eliminate the possibility of the risk following turn around solutions avoiding the threat.
- Risk prevention, which is the typical situation. Protection, mitigation and anticipation are the key factors to reduce risk.
- Risk transfer, which implies the shift of the consequences of the risk to another organization.

- Risk contingency, which implies the use of reserve resources to mitigate an actual threat according to a previously established contingency plan.

6. The proposed model for risk assessment

Transforming the unstructured problem of risk assessment leads to an objective method able to be translated into an algorithm. In order to structure the problem, we decompose risk assessment of an engineering project in two different visions. First, a micro-vision is required for threat resolution. This micro-vision risk assessment relates to the identification of the threats, the decision-making process to address the problem, and the formalization of the solution in a plan.

The micro vision is necessary but not sufficient because it is impossible to manage a project without a global scenario. Hence, a macro vision approach is also required. The macro vision approach relates to the integration of the evaluation made for each of the threats. The macro-vision risk assessment of the project includes three risk components: process, product and resources.

6.1. Micro-vision

The decision-maker is positioned on the root of a decision tree, where each branch represents a course of action that implies costs and probabilities of success. When a threat is identified, two possible choices are available: to avoid the threat or to deal with it. Avoiding a threat is usually associated with represent some costs. Typically, avoiding a threat implies finding a turn around that can have effects on schedule, budget or even on functionality.

If the decision-maker opts to deal with the threat, then three possible courses of action are available: to prevent, to wait, or to transfer the threat. Prevention and transfer could have associated costs. The waiting strategy postpones the use of resources in the hope that the threat will not appear, trying to trade information for time.

Even if applying prevention, there is no absolute guarantee that the threat will not appear. In these cases the decision-maker can apply a contingency plan that introduces new costs. Again the contingency plan cannot guarantee absolute effectiveness.

If we know or can estimate the probability of each branch representing a state of nature, it is possible to calculate the expected outcome for each one as the weighted sum of outcomes. So, we can arrive to the root with a value for the expected cost. The path that produces an optimal expected solution contains the recommended course of action.

To solve the uncertainties, subjective estimation of the probabilities of occurrence of the different states of nature can be applied. This approach is easy to implement but requires a great deal of experience to judge accurately the success probability of each alternative. Group consensus techniques (like the Delphi method) are usually very helpful in such situations.

Decisions trees based on the expected monetary value (EMV) could lead to bad decisions because in the most common case the decision-maker is confronted with a multiattribute problem. Moreover, different people have different attitudes toward risk. This issue is applying utility theory. The decision-maker must provide his estimation of return for each attribute related to the decision, as a vector R = (R1, R2, ..., Rn). The decision-maker must introduce also his preferences as a weight vector W = (W1, W2, ..., Wn). The outcomes of each attribute are given by Ai, such that:

$$Ai = Wi * Ri \qquad \text{, where } \sum_{i=0}^{n} Wi = 1$$

The outcome for each alternative is then calculated as a function of the sum of the attributes (A1, A2, ..., An) converted to a value between 0 and 1, where 1 is given to the best outcome and 0 to the worst.

6.2. Macro-vision

As we stated previously, the macro-vision approach integrates the assessments done for each of the identified threats. Moreover, the macro-vision risk can be used to find threats in an automated way. The risk assessment for the project is done by the integration of three risk factors (process, resources and product), plus two customization factors (decision-maker's perceptions of the environment and decision-maker's preferences).

The process introduces risk as consequence of its requirements and characteristics: complexity, technology required, budget required, schedule required, and personnel skills required. The process provides the description of its environment and the theoretical requirements to execute it.

The resources represent the actual allowances in personnel, tools, budget and schedule. The resources impose constraints that may not match the process requirements. These mismatches are a source for threats that can be identified automatically.

The product introduces its own risk in terms of quantitative and qualitative attributes. We identified two basic product-risk factors: requirement conflicts, and requirement complexity. The second one is consequence of the functional complexity of the requirements and the quality target defined in terms of reliability, maintainability and usefulness.

The risk assessment of the project can be structured as the evaluation of the complexities and the degree of mismatch from the product and process characteristics, to the resource constraints. The process of collecting risk metrics can be automated at least for the principal factors. Hence, project risk can be assessed using an automated tool.

7. Metrics

Metrics are a key factor in the identification of threats. Without metrics it is not possible to provide early alerts of risks. In this section we describe a set of metrics that support our risk identification strategy. All the metrics presented here are well formed, in the sense that they present the following strengths:

- Robust in terms of the verification of their outputs.
- Repeatable. Different observers would arrive at the same measurement regardless of the number of repetitions.
- Simple. We use the least number of parameters sufficient to obtain an accurate measurement.
- Easy to calculate. They do not require complex algorithms or processes.
- Automatically collected. There is no need of human intervention.

7.1 Metrics for Requirements

We define *birth rate* (BR) as the percentage of new requirements incorporated in each cycle of the evolution process. This metric shows the explosion of new requirements as a percentage.

BR % = (NR / TR) * 100, where

NR = number of new requirements,

TR = total number of requirements

TR = PR + NR, where PR denotes the number of requirements in the previous version.

We define *death rate* (DR) as the percentage of requirements that are dropped by the customer in each cycle of the evolution process.

DR % = (DelR / TR) * 100, where

DelR = number of requirements deleted,

TR = total number of requirements (before deletion) = PR + NR.

We define *change-rate* (CR) as the percentage of requirements changed from the previous version.

CR (%) = (ModR / TR) * 100

where ModR = number of requirements changed.

From the point of view of the metrics, a change on a requirement can be viewed as a death of the old version and a birth of the new one. This simplification does not imply that we lose the history of the evolution. The traceability of the evolution remains in the hypergraph model.

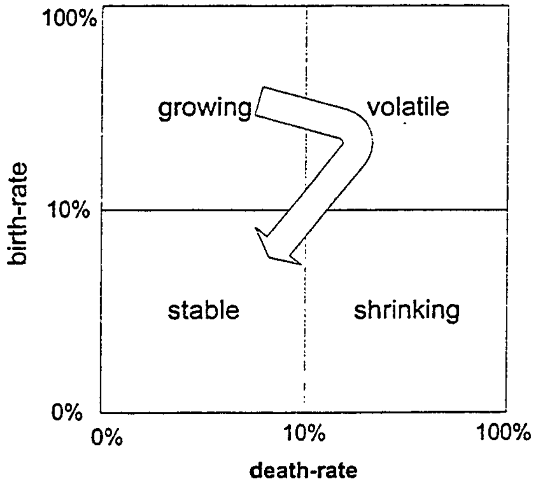


Figure 2: Evolution of requirements in a project

The simplification just described, enables us to compare birth rate and death rate in a two-dimensional plot that shows four regions: stability region, growing region, volatility region and shrinking region (fig. 2). The graph is double logarithmic, so the borders of the four regions are in the 10% value. Each of these regions has different risk connotations.

The arrow shows the normal evolution of a project as the time goes by. During early stages, it is normal for projects to be in the growing region. However, if the project continues in this region after many cycles, or return to this region after visiting other regions, something wrong is happening. The first case, this is an indicator that the requirement engineering is not efficient; hence some corrective action should be applied. The second case, shows evidence of late discovery of some cluster of hidden requirements.

After some cycles, the project should be in the volatile region. If the project does not evolve into the stability region, then there is evidence that the requirements engineering activity is not being efficient and some corrective action is mandatory. It is important to analyze the evolution of the stakeholder's issues and criticisms. It could be also the case that stakeholders have changed their minds.

If the project evolves to the shrinking region, and the requirements engineering is working properly, there is evidence that the customers are cutting down the project. This can be an indicator of a severe cut in the budget.

Finally, any involution to a previous region should be considered as evidence of threats. In such cases a detailed analysis is required to assess the causes of the anomaly.

This set of metrics can be collected automatically form the hypergraph and can give early alerts of the threats.

7.2 Metrics for Complexity

Complexity has a direct impact on quality because the likelihood that a component fails is directly related to its complexity. The quality of the product can only be determined at the end of the process. Hence, it is important to measure the complexity as predictor.

Real time systems present special difficulties in terms of requirement engineering. Some requirements are difficult for the user to provide and for the analysts difficult to determine. The best way to discover these hidden requirements is via prototyping. CAPS is a CASE tool specially suited for this task.

The prototyping process consists of prototype construction and modification (evolution) based on evolving requirements and code generation. Both construction and modification are exploratory activities with a common target: to satisfy multiple users with different and often conflicting points of view. Requirement engineering is a consensus driven activity in which mechanisms for conflict resolution and traceability of requirement evolution represent critical success factors.

Specifications written in PSDL, the prototyping language used in CAPS, are suitable for being analyzed to compute their complexity. In PSDL code we observe the following components: types, operators, data streams and constraints. Types are declarations of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. Finally, constraints represent guard conditions and real-time constraints that the system must support.

We define two complexity metrics for PSDL: Fine Granularity Complexity metric (FGC), and Large Granularity Complexity metric (LGC). The reason to compute different metrics is because we want to detect two classes of threats. First, we need to be aware of operators that are too complex. High complexity on one operator could be caused by poor design and possible can be solved by further decomposition. Second, we require a metric to compute the total complexity of the system.

FGC expresses the complexity of each operator in the system and is a function of the fan-in and fan-out data streams related to the operator.

$$FGC = \text{fan-in} + \text{fan-out}$$

LGC expresses the complexity of the system as a function of the number of operators, data streams, and types.

$$LGC = O + D + T$$

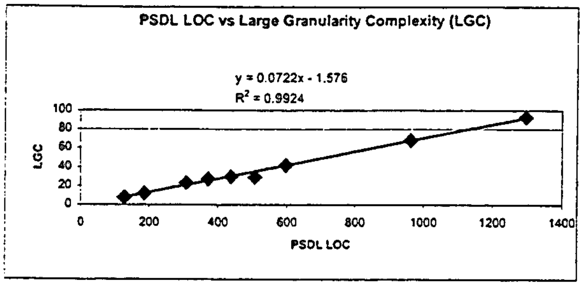


Figure 3: Correlation between PSDL and LGC

We examined the correlation between LGC and size of the specifications and the code. We observed a very strong correlation between PSDL lines of code and LGC (R = 0.996) (fig. 3). The correlation between non-comment Ada lines of code of the projects with their complexity measured using LGC, we observe a strong correlation also (R = 0.898) (fig. 4). Our complexity metric correlates better with PSDL than with Ada. The reason for this difference is because CAPS automatically generates PSDL. On the other hand, even if CAPS generates part of the Ada code, the designer can add and modify the generated code introducing more variability. The following graph shows the correlation observed for the same set of projects.

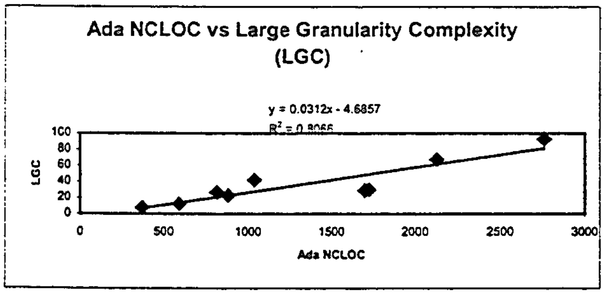


Figure 4: Correlation between NCLOC (Ada) and LGC

A caveat of this study is that our sample is too small. It includes all information we have available at the moment. However, the study suggests the possibility to estimate code size in terms of requirement complexity with useful levels of accuracy.

8. Integration with the graph model

The graph model has advantage of being easily expandable. The model is based on a hypergraph G = (N, E, I, O) where N is a set of nodes that represent the software components and related documents; E is a set of edges that represent the steps or tasks required by the process; I and O are functions that permit the navigation forward and backward in the graph. Risk assessment activities can easily be incorporated to the model by the extension of the class of edges. Figure 5 represents the software evolutionary prototyping software process. Figure 6 shows the proposed software process improvement. From the specifications we can derive the complexity of the product. This information is used together with personnel and organizational information, and with metrics of requirements collected from the baselines, to produce the risk assessment. The risk assessment step integrates these measures with issues created by the application of the REMAP model in the issue analysis steps. The automated risk assessment provides the decision-maker with objective and reliable information.

9. Conclusion

We introduced a framework and metrics able to structure the risk assessment problem and to solve it by automated tools. Further experiments should be conducted to validate our preliminary observations on complexity and size.

We found a method to solve the problem of human dependency in risk assessment. This method was designed for the graph model, however it can be customized to any evolutionary prototyping software process.

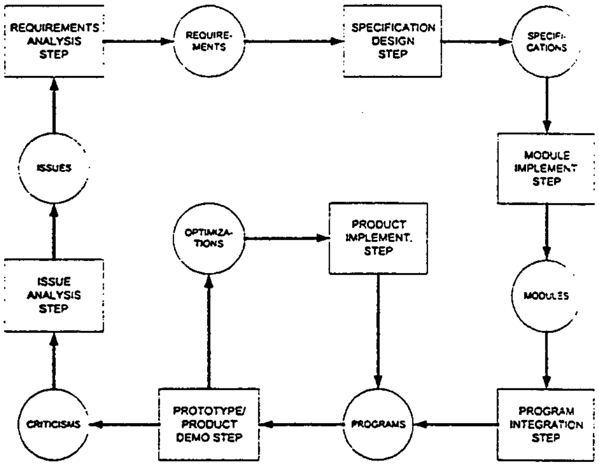


Figure 5: The evolutionary prototyping software process

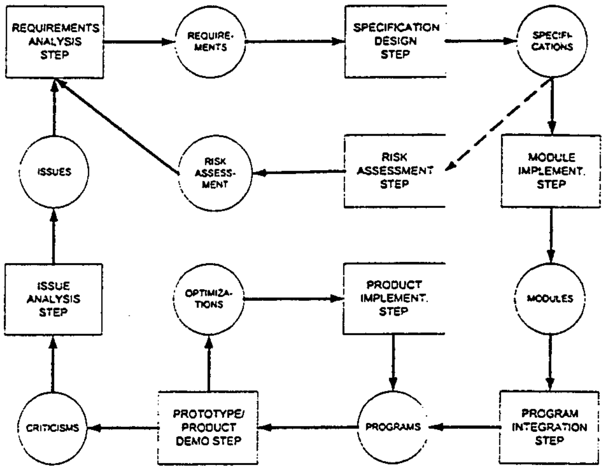


Figure 6: The proposed process

References

[1] Luqi. Software Evolution Through Rapid Prototyping. IEEE Computer. May, 1989.

[2] Ramesh, B. and Luqi. Process Knowledge Based Rapid Prototyping for Requirements Engineering. Journal of Systems Integration, 5 (157-177) 1995.

[3] Conklin, J. and Begeman, M. GIBIS: A Hypertext Tool for Exploratory Policy Discussion. ACM Transactions on Office Information Systems. Vol. 6. October, 1988.

[4] Luqi. A Graph Model for Software Evolution. IEEE Transactions on Software Engineering. Vol. 16 No. 8. August, 1990.

[5] Ibrahim, O. A Model and Decision Support Mechanism for Software Requirements Engineering. Ph.D. Dissertation. NPS. Monterey, California. 1996.

[6] Harn, M. Relayional Hypergraph Model. PhD Disssertation. NPS. Monterey, California. 1999.

[7] Mostov, Luqi and Hefner. A Graph Model of Software Maintenance. Technical Report NPS52-90-014. Department of Computer Science. NPS. Monterey, CA. August 1989.

[8] Badr, S. A Model and Algorithms for a Software Evolution Control System. PhD Dissertation, Computer Science Department. NPS. Monterey, CA. 1993.

[9] Luqi and Goguen, J. Formal Methods: Promises and Problems. IEEE Software. January, 1997.

[10] Boehm, B. A Spiral Model of Software Development and Enhancement. Computer. May, 1988.

[11] Hall, E. Managing Risk. Methods for Software Systems Development. Addison Wesley, 1997.

[12] Karolak, D. Software Engineering Management. IEEE Computer Society Press, 1996.

[13] Software Engineering Institute. Software Risk Management. Technical Report CMU/SEI-96-TR-012. June, 1996.

[14] Humphrey, W. Managing the Software Process. Addison-Wesley, 1989.

Surfing the Edge of Chaos: Applications to Software Engineering

Juan C. Nogueira
Carl Jones
Luqi
Naval Postgraduate School
2, University Circle
Monterey, CA. 93943 USA
+1 (831) 656 2093
jcnoguei@nps.navy.mil

Abstract

This paper discusses the problems of software engineering as the weakest link in the development of systems capable of achieving information superiority. Fast changes in technology introduce additional difficulties in terms of strategic planning, organizational structure, and engineering of software development projects. In such complex environment, a new way of thinking is required. We analyze the introduction of complex adaptive systems as an alternative for planning and change. The strategy of competition on the edge of chaos is analyzed showing the risks and the skills required navigating on the edge. We discuss the feasibility of using this theory in software engineering as an alternative to bureaucratic software development processes. We present also some recommendations that could help to acquire competitive advantage in software development, hence achieve information superiority.

1. Introduction

As software systems increased in complexity, software development evolved form a primitive art into software engineering. Methodologies and software tools were developed to help development processes. Most of the present tendencies (DOD-STD-2167A, ISO-9001, SEI/CMM) try to standardize processes, emphasizing planning and structure (Humphrey, 1990). Some authors criticize those approaches stating that they underestimate the dynamics of the software development (Bach, 1994), (Abdel-Hamid, 1997). Others question that activities such as research and development are not addressed by TQM principles (Dooley et al., 1994).

In 1994 Gibbs claimed "despite 50 years of progress, the software industry remains years—perhaps decades—short of the mature engineering discipline needed to meet the demands of an information-age society." Many researchers have treated the problem using different approaches: tools, formal methods, prototyping, software processes, etc. However, this assertion remains true today.

The typical software engineering process is a succession of decision problems trying to transform a set of fuzzy expectations into requirements, specifications, designs, and finally code and documentation. The traditional waterfall software process failed to accomplish their purpose because it applied a method valid for well-defined and quasi-static scenarios. This hypothesis is far from the reality. Today, modern software processes (Boehm, 1988), (Luqi, 1989) are based on evolution and prototyping. These approaches recognize the fact that software development presents an ill-defined decision problem and they fail in assessing automatically the risk.

In our view, software development projects present special characteristics that require to be solved in order to achieve an improvement in the state of the art. These particularities affect the strategic planning, the organizational structure, and the engineering applied to software. In these three areas chaos theory can provide clues for possible solutions.

2. The strategic planning issue

Traditional approaches to strategic planning emphasize picking a unique strategy according to the competitive advantages of each organization. Porter's five-force approach (Porter, 1980), assumes that there exists some degree of accuracy in the prediction of which industries and which strategic positions are viable and for how long.

In a high-velocity scenario the assumption of a stable environment is too restrictive. Customers, providers, competitors, and potential competitors, as well as substitute products are evolving faster than expected. The introduction of new information technology tools, the Internet and the globalization of the markets are contributing to this phenomenon, and nothing seems to reverse the process. The failure of long-term strategic planning is not a failure of management; it is the normal outcome in a complex and unpredictable environment. A growing number of consultants and academics (Santosus, 1998), (Brown & Eisenhardt, 1999) are looking at complexity theory, to help decision-makers improve the way they lead organizations.

How useful could a map of a territory that is constantly changing its topography be? In fast changing environments, survival requires a refined ability to sense the external variables. Traditional approaches rely on strategic planning and vision. However, in unstable environments planning would not be effective because it is impossible to predict the scenario's evolution in terms of markets, technologies, customer's needs, etc. Organizations relying only on one vision supported by a tight planning, risk paying little attention to the future. Consequently, their sensing organs are blind to foresight the future. A certain amount of inertia and commitment to the plans is required to prevent erratic changes caused by reaction diverse variables.

If the time window of the opportunities is shrinking, a different form of thinking is required. The present technological situation can be described as a fast succession of short-term niches. The ability to change is the key of success for surviving in such a variable environment. In a systemic approach, the General Systems Theory establishes that organizations are systems whose viability depends on some basic behaviors (von Bertalanfy, 1976):

(a) Ability to sense changes in the environment. This is the most primitive form of intelligence, if it is not present the probabilities of survive are minimum.

(b) Ability to adapt to a new environment modifying the internal structure and behavior. The system tries to auto-regulate to survive the crisis in hostile scenarios, or take advantage of the opportunities in favorable ones.

(c) Ability to learn from the past, anticipating the auto-regulation behaviors and structure before the environment change. This ability requires intelligence able to infer conclusions from the past according to the context of the variables sensed on the present.

(d) Ability to introduce changes in the environment, making it more favorable to the system's needs. In this case, the system has developed the technology (know how and tools) to exert power over the environment.

Any mechanical or computing system has some or all of these abilities. We find these same abilities in any form of life. The more developed the system is, the more of the above characteristics has. Darwin's Evolution Theory validates this line of reasoning. Natural selection, acting on inherited genetic variation through successive generations over the time is the form of evolution. Variation is the way used by biological systems to probe the environment presenting many alternatives, some of them ending on failure but a few very successful. This process is an inefficient but very effective way of improvement.

Experiments can provide a certain amount of knowledge about the future. In some sense, probes are mutations in small scale that can cause only small losses. The results give insights to discover new options to compete in the future and stimulate creative thinking. The research investment pays dividends when a new way of competition is discovered altering the status quo's rules.

When the changes in the environment occur too fast, sensing the variables becomes more difficult. It is possible that a specialized organ was not able to react on time to record the metric and transmit the alert. In this case, the system starts to lose information threatening its own viability. When the changes in the environment are too drastic, even if the sensor organs detect the change, the inference organs may not be able to determine an effective course of action because they do not have a previous experience, or because the decision-making process requires more time. This situation also threats the viability of the system in the long run. The effects of drastic variations and high rate of change over systems can be visualized with simple experiments: a) increasing the speed of transmission in a communication channel beyond some limit will provoke the lost of part or the entire message, b) modifying the pH in the soil beyond a certain limit can cause the death of a plant.

The same syndrome can be recognized in any type of organization. We purpose to employ a new strategy. "Competing on the Edge" is a new theory defines strategy as the creation of a relentless flow of competitive advantages that, taken together, form a semi-coherent strategic direction (Brown & Eisenhardt, 1999). The key driver for superior performance is the ability to change, reinventing the organization constantly over the time. This factor of success can be applied to software engineering as well as to other decision problems with similar characteristics.

If the environment is moving, like in surfing, the best way to remain in equilibrium is by being in the rhythm. Successful corporations such as Intel or Microsoft are in perpetual movement, launching new products with certain rhythm. Intel is faithful to its founder's (Moore) law: the power of the microprocessors double every eighteen months. Microsoft has a proportional pace on the software sector.

3. The organizational issue

The second unresolved issue is organizational. We think that many of the problems on current software projects have organizational roots. This opinion is also supported by (van Genutchen, 1991)[1] and (Capers Jones, 1994)[2].

Perrow (Burton et al., 1998), introduced a two-dimensional classification of the technology (Fig. 1). The first dimension is the analyzability of the problem varying from well defined to ill defined. The second dimension is the task variability, which means the number of expected exceptions in the tasks.

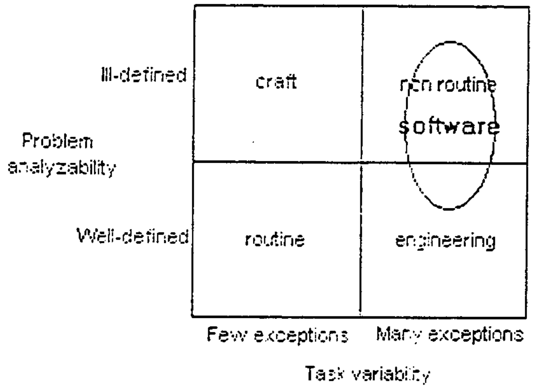


Figure 1: Perrow's classification of technology

In our view, a third dimension is required to model the dynamics of the problem. In general, any technological scenario will change its analyzability and its variability with time. This is the case for software engineering developments. During the initial stages the problem is ill-defined and many exceptions occur. After several evolution cycles, usually comprising several prototypes, the requirements become clear and the problem drift gradually into the engineering quadrant. In figure 1, the gray oval represents the projection of the software problems in a two dimensional space.

This kind of scenarios require highly skilled personnel, low formalization and centralization, high information processing demand, and coordination obtained through meetings is required. In our opinion software engineering is not the only discipline in this quadrant. The challenges imposed by hyper competition create similar characteristics than in software engineering developments. So, the rules of engagement proved effective for one discipline could result useful in the other.

A second line of research (Burton & Obel, 1998), introduced a classification based on four-variable model: equivocality, environmental complexity, uncertainty and hostility. Equivocality is *"the existence of multiple and conflicting interpretations"*, it is a measure of the lack of knowledge or the level of ignorance whether a variable exists in the space. Uncertainty is the lack of knowledge about the likelihood of values for the known variables. Environmental complexity is the number of factors in the environment affecting the organization and their interdependency. Finally, hostility *is "the level of competition and how malevolent the environment is."*

In Table 1, we disregard the fourth variable: hostility. Hostility is a discontinuity of the environment. When it is high, then it overrules other factors. In highly hostility scenarios only a highly centralized organization ("regular army"), or a low-formal-low-complex organization ("guerilla") are the possible alternatives.

[1] Van Genuchten found that 45% of all the causes for delayed software are related to organizational issues.

[2] Capers Jones found that on military software developments the two more common threats are excessive paperwork (90% of the time) and low productivity (85% of the time).

Software development scenarios usually correspond to high equivocality, high environmental complexity and high uncertainty scenarios (dark gray in the matrix), which correspond to low formalization and low organizational complexity, with centralization inverse to the environmental complexity. The recommended organization could be ad hoc or matrix with coordination by integrator or group meeting. The information exchange is rich and abundant. The incentive policy should be based on results.

Equivocality	Enviromental Complexity	Uncertainty	Formalization	Organizational Complexity	Centralization
Low	Low	Low	High	Medium	High
Low	Low	High	Medium	High	Medium
Low	High	Low	High	Medium	Medium
Low	High	High	Medium	High	Low
High	Low	Low	Medium	Medium	High
High	Low	High	Low	Low	High
High	High	Low	Medium	Medium	Low
High	High	High	Low	Low	Low

Table 1: Burton & Obel classification

Understanding these organizational characteristics inherent of software projects is required to create a more fitted software process. The application of a quasi-chaotic process keeps the organization in continuous movement with positive effects its internal behavior. The rhythmic change avoids manager's tendency to slow down the process or introduce changes too often. The periodic changes create small amounts of chaos that maintain the organization in the edge.

4. The engineering issues

Despite 50 years of progress, the software industry remains immature to meet the demands of an information-age economy. Many researches have treated the problem using different approaches: formal methods, prototyping, software processes, etc. However, the problem remains open today. The third unresolved issue is a set of engineering problems concerning software processes, risk assessment, and reuse.

4.1. The software process problem

Studies have shown that early parts of the system development cycle such as requirements and design specifications are especially prone to error (Luqi, 1989). Problems originating in the early stages often have a lasting influence on the reliability, safety and cost of the system. This effect is particularly notorious in projects involving multiple stakeholders with different points of view. Evolutionary software processes offer an iterative approach to requirement engineering to alleviate the problems of uncertainty, ambiguity and inconsistency inherent in software developments. Experience suggests that building and integrating software by mechanically processable formal models leads to cheaper, faster and more reliable products. Moreover, prototyping can improve the capture of change in requirements and assumptions during the development process. Prototypes are useful to demonstrate system scenarios to the affected parties as a way to: a) collect criticisms and feedback that are sources for new requirements; b) enable early detection of deviations from users' expectations; c) trace the evolution of the requirements; and d) improve the communication and integration of the users and the development personnel.

Despite the unquestionable benefits of evolutionary software processes, we have some concerns. The first concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product. Most project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.

Second, evolutionary software processes do not establish the maximum speed of the evolution. If the evolutions occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then the productivity could result affected. The correct rhythm for software processes has not been researched and remains on the hands of the project manager.

Third, software processes should be focused on *flexibility* and *extensibility* rather than in *high quality*. This assertion sounds scary. However, we should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product, when the opportunity niche has disappeared. This paradigm shift is imposed by the competition on the edge of chaos.

4.2. The risk assessment and estimation problems

Developing software is still a high-risk activity. Despite the advances in technology and tools, little progress has been done in improving the management of software development projects. Part of the problem is misinterpretation of the importance of risk management that is usually viewed as an extra activity layered on the assigned work, or worst, as an outside activity that is not part of the software process (Hall, 1997), (Karolak, 1996).

Software development processes such the hypergraph model for software evolution (Luqi, 1989), or the spiral model (Boehm, 1988), improved the state of the art. However, all of them have a common weakness: risk assessment.

On the software evolution domain, risk assessment has not been addressed as part of the model. In the various enhancements and extensions, the graph model did not include risk assessment steps; hence risk management remains as a human-dependent activity that requires expertise.

On the evaluation of the spiral model, one of the difficulties mentioned by Boehm was: *"Relying on risk-assessment expertise, the spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk."* (Boehm, 1988).

Many researches have addressed the problem of risk assessment following only one perspective. The available tools for risk assessment are guidelines for practices, checklists, taxonomies of risk factors and few metrics. All these methods work fine if a) there is a human educated on risk assessment, and b) he/she has enough experience. Such resources are very scarce and it is difficult to leverage their expertise over large organizations.

The main line of previous research has addressed the problem in parallel with the development process using informal methods. Basically the proposed methodologies are lists of practices and checklists (SEI, 1996), (Hall, 1997) or scoring techniques (Karolak, 1996) that are dependent on human expertise.

The second weakness on risk assessment is caused by the difficulties in estimate the development tiem. The industry has been using three classes of tools to estimate effort and time that can be applied at different moments during the life cycle, each category being more precise than the previous one but arriving later on the life cycle:

a) Very early estimations. This category includes very crude approximations done during the beginning of the process usually by subjective comparisons using previous projects.

b) Macro models. This category includes Basic COCOMO, COCOMO II (application composition model), Putnam, Function Points, etc. The estimation is done after completing the requirements phase.

c) Micro models. This category includes intermediate and detailed COCOMO, COCOMO II (early design and post-architecture models), and Pert/CPM/Gantt techniques. The estimation is done after the design when it is possible to have a work breakdown structure. The project estimate is the integration of all module estimates.

A detailed discussion of these techniques is outside the scope of this paper; the details can be read in (Albrecht, 1979 and 1983), (Boehm, 1981 and 2000), (Londeix, 1987), (Putnam, 1980, 1992, 1996, and 1997). None of these techniques consider the following characteristics of software projects:

a) Requirement volatility

b) Personnel volatility

c) Time consumed by communications, exceptions and noise in the process. All the methods use size as an input parameter via some kind of derivation from complexity. In many cases the methods to compute such complexities and sizes are questionable (Kitchenham, 1993 and 1997), (Kemerer, 1993).

Recently, NPS developed a formal model for risk identification and assessment for evolutionary software processes that solves the problems of automation, human dependency, and estimation (Nogueira et al. 2000). This research is focused on studying software project risk assessment from a different perspective, viewing risk assessment as the prediction of success of the project given a set of characteristics, a probabilistic model based on Weibull distribution, and learning from each successive cycle on the process.

4.3. *The reuse problem*

Even if the industry claims for the use of flexible and extensible architectures from which reusable components could be integrated as a way of generating applications, the reality is that the standard does not exist. Different architectures are competing for becoming the de facto standard. Microsoft proposes the Distributed network Architecture (DNA) based on DCOM and ActiveX. Sun and other OMG members propose the Enterprise Computing Platform (ECP) based on IIOP and CORBA. Each alternative presents advantages and disadvantages and it is not easy to forecast the winner.

5. The edge of chaos

The edge of chaos is *"a natural state between order and chaos, a grand compromise between structure and surprise"* (Kauffman, 1995). Chaos theory describes a specific range of irregular behaviors in systems that move or change (James, 1996). Chaotic does not mean random. The primary feature distinguishing chaotic from random behavior is the existence of one ore more attractors. Without the existence of such attractors the quasi-chaotic scenarios could not be repeatable. It is important to realize that a chaotic system must be bounded, nonlinear, non-periodic and sensitive to small disturbances and mixing. If a system has all these properties can be driven into chaos.

We have the tendency to think that the order is the ideal state of nature. This could be a big mistake. Research on organizational theory (Stacey, Nonaka, Zimmerman); management (Stacey, Levy); and economics (Arthur) support the theory that operation away from equilibrium generates creativity, self-organization processes and increasing returns (Roos, 1996). Absolute order means the absence of variability; consequently this behavior could be very dangerous in environments with high equivocality. In such scenarios, a better approach could be a restless series of changes aiming competitive advantage niches, which globally form a semi-coherent strategic direction.

Change occurs when there is some structure so that the change can be organized, but not so rigid that it cannot occur. Too much chaos, on the other hand, can make impossible the coordination and coherence. Lack of structure does not always mean disorder. Let illustrate this idea with an example. We can agree that there is little structure in a flock of migratory ducks in a lake. However, few minutes after they start flying some order appear and the flock creates a V shape formation. This self-organized behavior occurs because a loose form of structure exists. Experiments with intelligent agents governed by three rules (a) try to maintain a minimum distance from the other objects in the environment, including other agents; b) try to match the speed of other agents in the vicinity; and c) try to move toward the perceived center of mass of the agents in the vicinity), show the same behavior. Independently of the starting position of the agents, they always end up in a flock. Even if an obstacle disturbs the formation, the pseudo-order is recovered some time later. This self-organized behavior emerges despite the absence of leadership and without an explicit order to form a flock.

A more interesting example is the behavior of software development teams. A recent article (Cusumano, 1997), describes the strategies of Microsoft to manage large teams as small teams. Dr. Cusumano says "*What Microsoft tries to do is allow many small teams and individuals enough freedom to work in parallel yet still function as one large team, so they can build large-scale products relatively quickly and cheaply. The teams adhere to a few rigid rules that enforce a high degree of coordination and communication.*" This is an exact description of the emerging behavior in a complex adaptive system. It is self-adaptive because the agents realize the adjustment to the environment, and it is emergent because it arises from the system and can only be partly predicted. As in the example of the ducks, few rules of interaction between the agents (in this case people) generate a performing behavior. The three rigid rules at Microsoft are: a) developers integrate their work daily forcing the synchronization and testing of the work; b) developers responsible for bugs must fix them immediately, and are responsible for the next day integration; and c) milestone stabilization is sacred.

Complex adaptive systems, as the one just described, are made up with multiple interacting agents. The emergence of the complex behavior requires three conditions. First, it is required the existence of more than one agent. Second, the agents must be sufficiently different to each other such that their behavior is not exactly the same in all cases. When agents behave exactly the same way exhibit predictable, not complex, behavior. Finally, complex adaptive behavior only occurs in the edge of chaos.

6. Some of the risks of being in the edge of chaos

Limiting the structure in organizations can be useful in situations when innovation is critical or when is required to revitalize bureaucracies. However, if the structure is debilitated beyond a certain minimum, it can conduct to an undesired state. Some traits can alert the eminence of such anarchic situation known as the "chaos trap" (Brown & Eisenhardt, 1999): a) emerging of a rule-breaking culture, b) missing deadlines and unclear responsibilities and goals, and c) random communication flows.

On the other hand focusing in hierarchy and disciplined processes, emphasis on schedules, planning and job descriptions may conduct to a steady inert bureaucracy. Organizations in such state react too late failing to capture shifting strategic opportunities. This is the case of a "bureaucratic trap", where there are also some observable warning traits: a) rule-following culture, b) rigid structure, tight processes and job definitions, and c) formal communication as the only channel.

The alternative is "surfing" the edge of chaos avoiding both attractors. That requires limited structure combined with intense interaction between the agents, giving enough flexibility to develop surprising and adaptive behavior. Organizations in this state are characterized by having an adaptive culture. People expect and anticipate changes. A second characteristic is that the few key existing structures are never violated. Finally, real time communication is required throughout the entire organization.

Being in the edge of the chaos implies an unstable position. Some perturbations can cause the rupture of this delicate equilibrium and the fall into one of the two steady states. A potential perturbation factor is the organizational collaboration style. Too much collaboration can disturb the performance of each agent and consequently, the whole system is affected. On the other hand, too little collaboration destroys the advantage of acting organized and leads to paralysis.

Another sources of perturbation are the tendency to be tight to the past and cultural idiosyncrasy, or by contrary, to loose the link with the past. In one case, the change becomes impossible. In the other case, the assets from previous experiences are not capitalized. The equilibrium point is called regeneration. In such unstable state, mutation can occur. Therefore the inherited characteristics that give competitive advantage in a certain scenario can be perpetuated, and new variations are introduced. If too little variation exists, natural selection fails. This process permits that complex adaptive systems change over the time following a Darwinian pattern.

(Kauffman, 1995) introduced the concept of fitness landscape. We can understand this concept observing the behavior of species. In the competition for survival, species attempt to alter their genetic make-up by taking adaptation trying to move to higher "fitness points" where their viability will be enhanced. Species that are not able to reach higher points on their landscapes may be outpaced by competitors who are more successful in doing so. If that occurs the risk of extinction increases. The same principle applies between predator and prey. Each development in the abilities of one species generates an improvement on the abilities of the other. This concept is called co-evolution.

Certain higher fitness points have more value to some species than to others. The contribution a new gene can make to a species' fitness depends on genes the species already has. As more complicated is the genetic pattern (more evolved), the probability of conflict of a new adaptation increases slowing down the speed of variations.

Natural selection is an effective, but not generally efficient way to evolve. The process requires some amount of mutation to avoid the sudden convergence on suboptimal characteristics. Some of the characteristics lost in the past can be reintroduced being useful in the new scenario. Many errors are committed during this blind process. A more efficient way to evolve is by recombination of the pool of genes using genetic algorithms. This technique has been applied to improve the performance of robots, however the idea can be used to improve the competencies of organizations. If too much or too less variation occurs the result always conduct to the failure of the system.

7. Application in software engineering

Chaos in software development comes from various sources: a) the intrinsic variable nature of requirements, b) the changes introduced by new technologies, c) the dynamics of the software process, and d) the complex nature of human interaction. These non-linear characteristics plus the condition of edge of chaos are sufficient for the development of complex adaptive systems in which the agents are collaborative developer teams.

In software development scenarios equivocality, environmental complexity and uncertainty are usually high. The suggested organizational structure to deal with such scenarios (Burton & Obel, 1998) should have low formalization and organizational complexity, centralization inverse to the environmental complexity, and rich and abundant information exchange. The recommended organization should be ad hoc or matrix, with coordination by integrator or group meeting. This organizational style is difficult to achieve when the organizations are large. A clear solution to this problem was recognized at Microsoft (Cusumano, 1997): a) parallel developments by small teams with continuous synchronization and periodically stabilization, b) software evolution processes where the product acquires new features in increments as the project proceeds rather than at the end of a project, c) testing conducted in parallel as part of the evolution process, and d) focus creativity by evolving features and "fixing" resources. Cusumano observed that small development teams were more productive because: a) fewer people on a team have better communication and consistency of ideas than large teams, and b) in research, engineering and intellectual work individual productivity has big variance. Software development requires teamwork, more specifically organized work. So we require understanding the dynamics of organizations as artificial social entities that exist to achieve a specific purpose, in this case to develop software. Such organizations are made up of individuals who accomplish diverse desegregate activities that require coordination and consequently information exchange.

A shift from the traditional long-term development organizations is required. Virtual teams created as temporary dynamic project-oriented structures, with a composition of skills matching exactly the objectives could improve the current performances. Such virtual organizations are not exposed to bureaucratic loads and do not require to absorb the cost of permanent staff (Senegupta & Jones. 1999).

Larger developments could be achieved by parallel projects loosely coupled sharing a common architecture such CORBA or DCOM. This paradigm enables the possibility of managing large developing organizations as if they were small. In such scenarios, the benefits of complex adaptive systems will occur at two levels. First at the micro level, that is inside each small project, where the agents are individuals. Second, at the macro level, where the agents are parallel collaborative projects.

8. Conclusion

Complex adaptive systems appear as the most attractive way to deal with changing environments. Besides some indicators introduced by (Brown & Eisenhardt, 1999), the academic research is not mature enough to assert a methodology for competition on the edge. Some enterprises, such as Microsoft and Intel, seem to have discovered and applied this form of strategy since many years ago, but little information have permeated.

We propose a drastic change in the software processes using the benefits of programming in the small to programming in the large. More even, we state the quality-driven paradigm should be revised. and that the objective should be shorter delivery times, flexibility and expansibility.

Despite the obvious differences in terms of hostility, we found several similarities between war and software development scenarios. A depth research is required to evaluate the applicability of this theory to different fields in which uncertainty is a key factor peace keeping operations, joint C^{4}I, and irregular warfare.

References

(Abdel-Hamid, 1997) Abdel-Hamid. T. Lessons Learned from Modeling the Dynamics of Software Development. Edited by Kemerer, C. McGraw Hill 1997.

(Albrecht, 1979) Albrecht, A. Measuring Application Development Productivity. Proceedings IBM. October 1979.

(Albrecht, 1983) Albrecht, A. and Gaffney, J. Software Function Source Lines of Code and Development Effort Prediction. IEEE Transactions Software Engineering. SE-9, 1983.

(Bach, 1994) Bach, J. The Immaturity of the CMM. American Programmer, September 1994.

(Boehm, 1981) Boehm, B. Software Engineering Economics. Prentice Hall, 1981.

(Boehm, 1988) Boehm. B. A Spiral Model of Software Development and Enhancement. Computer. May, 1988.

(Brown & Eisenhardt, 1999) Brown, S. and Eisenhardt, K. Competing on the Edge. Strategy as Structured Chaos. Harvard Business School Press, 1999.

(Burton & Obel, 1998) Burton, R. and Obel. B. Strategic Organizational Diagnosis ans Design. Developing Theory for Application. Second Edition. Kluwer Academic Publishers, 1998.

(Cusumano, 1997) Cusumano, Michael How Microsoft Makes large Teams Work Like Small Teams. Sloan Management Review. Fall, 1997.

(Dooley, 1994) Dooley, K. and Flor, R. "Success and Failure in Total Quality Management Initiatives", Proceeding of the Chaos Network, Denver, 1994.

(Hall, 1997) Hall, E. Managing Risk. Methods for Software Systems Development. Addison Wesley, 1997.

(Humphrey, 1990) Humphrey, Watts. Managing the Software Process. Addison-Wesley, 1990.

(James, 1996) James, G. E. Chaos Theory. The Essentials for Military Applications. Naval War College. The Newport Papers. 1996.

(Karolak, 1996) Karolak, D. Software Engineering Management. IEEE Computer Society Press, 1996.

(Kauffman, 1995) Kauffman, Stuart. At Home in the Universe. Oxford University Press, 1995.

(Kemerer, 1993) Kemerer, C. Reliability of Function Points Measurements: A Field Experiment. Communications of ACM, Vol 36 No 2. 1993.

(Kitchenham, 1993) Kitchenham, B., Kansala. K. Inter-item Correlations among Function Points. First International Software metrics Symposium. IEEE Computer Society Press. 1993.

(Kitchenham, 1997) Kitchenham, B., Linkman. S. Estimates, Uncertainty, and Risk. IEEE Software. May-June. 1997.

(Londeix, 1987) Londeix, B. Cost Estimation for Software Development. Addison-Wesley, 1987.

(Luqi, 1989) Luqi. Software Evolution Through Rapid Prototyping. IEEE Computer. May, 1989.

(Nogueira et al., 2000) Nogueira, J.C., Luqi, and Berzins. V. A Formal Risk Assessment Model for Software Evolution. Paper submitted to SEKE 2000.

(Porter, 1980) Porter, Michael. Competitive Strategy. Free Press, 1980.

(Putnam, 1980) Putnam, L. Software Cost Estimating and Life-cycle Control: Getting the Software Numbers. IEEE Computer Society Press. 1980.

(Putnam, 1992) Putnam, L. and Myers, W. Measures for Excellence. Reliable

Software On Time Within Budget. Yourdon Press, 1992.

(Putnam, 1996) Putnam, L. and Myers, W. Executive Briefing. Controlling Software Development. IEEE Computer Society Press. 1996.

(Putnam, 1997) Putnam, L. and Myers, W. Industrial Strength Software. Effective Management Using Measurement. IEEE Computer Society Press, 1997.

(Roos, 1996) Roos, Johan. The Poised Organization: Navigating Effectively on Knowledge Landscapes, 1996. http://www.imd.ch/fac/roos/paper_po.html

(Santosus, 1998) Santosus, Megan. Simple, Yet Complex. Business Management CIO Enterprise Magazine. April 15, 1998.

(SEI, 1996) Software Engineering Institute. Software Risk Management. Technical Report CMU/SEI-96-TR-012. June, 1996.

(Senegupta & Jones, 1999) Sengupta, K. and Jones Carl R. Creating Structures for Network-Centric Warefare: Perspectives from Organizational Theory. Command & Control Researc & Technology Symposium. CCRP 1999. Naval War College, 1999.

(von Bertalanfy, 1976) von Bertalanfy, L. General System Theory: Foundations, Development. Applications. Braziller. 1976.

A Formal Risk Assessment Model for Software Evolution*

Juan C. Nogueira
Luqi
Valdis Berzins
Nader Nada

Naval Postgraduate School
2, University Circle.
Monterey, CA. 93943 USA
+1 833 656 2093
jcnoguei@nps.navy.mil

ABSTRACT

The current state of the art techniques of risk assessment rely on checklists and human expertise. This constitutes a weak approach because different people could arrive at different conclusions from the same scenario. The difficulty on estimating the duration of projects applying evolutionary software processes contributes to add intricacy to the risk assessment problem. This paper introduces a formal method to assess the risk and the duration of software projects automatically. The method has been designed according the characteristics of evolutionary software processes. We introduce a set of metrics to measure productivity, requirement volatility and complexity. We construct a formal method based on these three indicators to estimate the duration and risk of evolutionary software processes. The approach introduces benefits in two fields: a) automation of risk assessment and, b) early estimation method for evolutionary software processes.

Keywords

Risk, software metrics, estimation models

INTRODUCTION

Despite progress in formal methods, prototyping, and evolutionary software processes, risk assessment remains as an open issue dependent on human expertise. Software development processes such the hypergraph model for software evolution [15], or the spiral model [3], have a common weakness: risk assessment. In the software evolution domain, risk assessment has not been addressed as part of the model. In the various enhancements and extensions, the graph model did not include risk assessment steps, hence risk management remains as a human-dependent activity that requires expertise. In the evaluation of the spiral model, one of the difficulties mentioned by Boehm was: *"Relying on risk-assessment expertise, the spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk."* [3].

Many researches [9, 6, 20] have addressed the problem of risk assessment following guidelines, checklists, taxonomies of risk factors, and few metrics. All these methods work fine if a) they are applied by a human educated on risk assessment, and b) he/she has enough experience. The weakness of all current risk assessment practices is human dependency. As a corollary, risk assessment could not be consistent because different experts could arrive at different conclusions from the same scenario.

Our research is focused on transforming the present state of the art about risk assessment into a formal method. This paper introduces an automated and formal software project risk assessment model, based on early metrics and probabilities designed for evolutionary software processes.

THE PROBLEM

Studies have shown that early parts of the system development cycle such as requirements and design specifications are especially prone to error [15]. Problems originating in the early stages often have a lasting influence on the reliability, safety and cost of the system. This effect is particularly notorious in projects involving multiple stakeholders with different points of view. Evolutionary software processes offer an iterative approach to requirement engineering to alleviate the problems of uncertainty, ambiguity and inconsistency inherent in software developments. Moreover, prototyping can improve the capture of change in requirements and assumptions during the development process. Prototypes are useful to demonstrate system scenarios to the affected parties as a way to: a) collect criticisms and feedback that are sources for new requirements; b) enable early detection of deviations from users' expectations; c) trace the evolution of the requirements; and d) improve the communication and integration of the users and the development personnel.

Despite the unquestionable benefits of evolutionary software processes, we have two concerns. First, the automated risk assessment issue has not been resolved. It is usually viewed as an extra activity layered on the assigned work, or worst, as an outside activity that is not part of the software process [6, 9]. The main line of

* This research was supported by the US Army Research Office under grant #38690-MA and grant #40473-MA.

previous research has addressed the problem in parallel with the development process using informal methods. Basically the proposed methodologies are lists of practices and checklists [20, 6] or scoring techniques [9] that are dependent on human expertise.

The second concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product. The industry has been using three classes of tools to estimate effort and time that can be applied at different moments during the life cycle, each category being more precise than the previous one but arriving later:

a) Very early estimations. This category includes very crude approximations done during the beginning of the process usually by subjective comparisons using previous projects.

b) Macro models. This category includes Basic COCOMO, COCOMO II (application composition model), Putnam, Function Points, etc. The estimation is done after completing the requirements phase.

c) Micro models. This category includes intermediate and detailed COCOMO, COCOMO II (early design and post-architecture models), and Pert/CPM/Gantt techniques. The estimation is done after the design when it is possible to have a work breakdown structure. The project estimate is the integration of all module estimates based on linear layouts of activities, so they do not fit completely with evolutionary software processes.

A detailed discussion of these techniques is outside the scope of this paper; the details can be read in [1, 2, 4, 6, 14, 16, 17, 18, 19]. None of these techniques consider the following characteristics of software projects: a) requirement volatility, b) personnel volatility, and c) time consumed by communications, exceptions and noise in the process. All the methods use size as an input parameter via some kind of derivation from complexity. In many cases the methods to compute such complexities and sizes are questionable [10, 11, 12].

METRICS

In this section we describe a small set of metrics that support our risk identification strategy (requirements, personnel and complexity). We choose metrics presenting the following characteristics: a) robustness, b) repeatability, c) simplicity in terms of the number of parameters, d) easy to calculate, and e) automatically collectable.

Metrics for requirements

We purpose three metrics for requirements: a) birth-rate, b) death-rate, and c) change-rate. We define *birth-rate* (BR) as the percentage of new requirements incorporated in each cycle of the evolution process. This metric shows the introduction of new requirements as a percentage.

We define *death-rate* (DR) as the percentage of requirements that are dropped by the customer in each cycle of the evolution process.

We define *change-rate* (CR) as the percentage of requirements changed from the previous cycle.

From the point of view of the metrics, a change in a requirement can be viewed as a death of the old version and a birth of the new one. The simplification just described enables comparison of birth-rate and death-rate in a bi-dimensional plot that shows four regions: stability region, growing region, volatility region and shrinking region (fig. 1). Each of these regions has different risk connotations. The arrow shows the normal evolution of a project as time goes by. During early stages, it is normal for projects to be in the growing region. However, if the project remains in this region after many cycles, or returns to this region after visiting other regions, something wrong happens. The first case is an indicator that the requirement engineering is not efficient; hence some corrective action should be applied. The second case shows evidence of late discovery of some cluster of hidden requirements.

After some cycles, the project should be in the volatile region. If the project does not evolve into the stability region, then there is evidence that the requirements engineering activity is not efficient and some corrective action may be needed. It is important to analyze the evolution of the stakeholders' issues and criticisms. It could be also the case that stakeholders have changed their minds. If the project evolves to the shrinking region, and the requirements engineering is working right, there is evidence that the customers are cutting down the project. This can be an indicator of a severe cut in the budget. Finally, any return to a previous region should be considered as evidence of threats. In such cases a detailed analysis is required to assess the causes of the anomaly. This set of metrics can be collected automatically from the baseline and can give early alerts of threats. In our schema, requirement volatility is related to two risk factors: the product and the process.

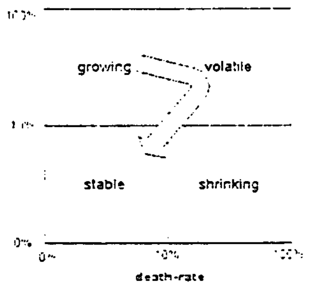


Figure 1: Evolution of requirements

Metrics for fitness

We require measure the fit between people and their roles in the software process. In order to measure personnel both quantitative and qualitative metrics are required. A skill match between person and job is required to estimate the speed in processing information and rate of exceptions. On the quantitative side it is important to measure the number of people and the turnover. This last one provides information about the expected productivity losses due to training, learning curves and communications. This set of metrics is difficult to collect because people are very reluctant to being measured. During the simulations we found that there exists an easier way to measure the productivity fitness observing the ratio between direct working time and idle time as we will discuss in 6.1. Fitness is related to two risk factors: the resources and the process.

Metrics for complexity

Complexity has a direct impact on quality because the likelihood that a component fails is directly related to its complexity. The quality of the product can only be determined at the end of the process. Hence, it is important to measure the complexity as an early predictor to provide a way to assess the duration of the project given some indicators collected during the requirements phase. In such conditions, code is not available, so the only possible measurements should come from the specification. Complexity is related to one risk factor: the product.

Research on Function Points (FP) [1, 2] showed that there exists a clear relation between complexity and size in terms of lines of code. However, FP are not well suited for real time systems or object-oriented developments [10, 11, 12].

Formal specifications are suitable for being analyzed to compute their complexity. We conducted experiments trying to derive complexity from formal specifications created by CAPS (Computer Aided Prototyping System) [15]. The tool generates specifications in a structured language called Prototyping Specification Design Language (PSDL). PSDL code has the following components: types, operators, data streams and constraints. Types are declarations of abstract data types required for the system. Operators are state machines and data streams represent the communication links between them. Both operators and data streams are the components of a dataflow graph. Finally, constraints represent the real-time constraints that the system must support. The tool generates Ada code form PSDL specifications.

We defined two complexity metrics for PSDL: a) *Fine Granularity Complexity* metric (FGC), and b) *Large Granularity Complexity* metric (LGC). The reason to compute different metrics is because we want to detect two classes of threats. First, we need to be aware of excessively complex operators. High complexity of one operator could be caused by poor design and possibly can be solved by further decomposition. Second, we require a metric to compute the total complexity of the system.

FGC expresses the complexity of each operator in the system and is the sum of the fan-in and fan-out data streams related to the operator (FGC = fan-in + fan-out).

LGC expresses the complexity of the system as a function of the number of operators (O), data streams (D), and types (T) (LGC = O + D + T).

We found a strong correlation between PSDL lines of code and LGC (R = 0.996, fig. 2). If we compare the Ada non-comment lines of code of the projects with their complexity measured using LGC, we observe a strong correlation also (R = 0.898, fig.3). Our complexity metric correlates better with PSDL than with Ada because CAPS automatically generates PSDL; on the other hand, even if CAPS generates part of the Ada code, the designer can add and modify the generated code, introducing more variability. The size of the project in thousands of non-comment lines of code can be estimated as:

KLOC = (32 LGC − 150) 1000 [Eq. 1]

As the complexity grows, the ratio trends to approximately 32 LOC for each unit of LGC. This finding provided us with a method to compute the size of the projects given an early measure of their complexity. This conversion is required to compare our approach with Putnam's and Boehm's approaches because they require the size as an input parameter. A caveat of this study is that our sample is small, but it includes all the information we have at the current time. However, the study suggests the possibility of estimating size in terms of complexity with a useful degree of accuracy.

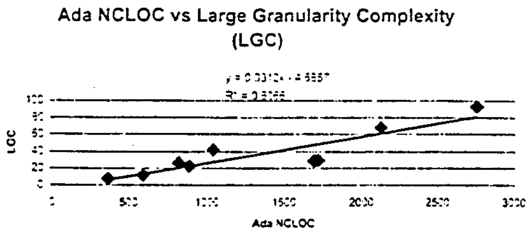


Figure 2: Correlation between PSDL and LGC

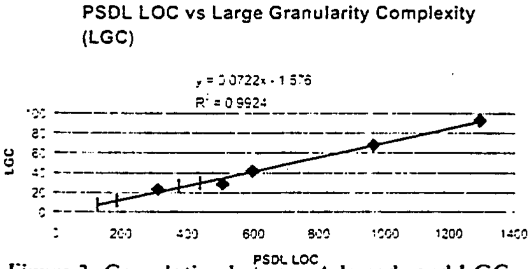


Figure 3: Correlation between Ada code and LGC

THE RISK ASSESSMENT MODEL

A probability distribution from the Weibull family can be used to model the development time given the risk factors discussed above. The probability density function and cumulative density function for the model are:

$$\text{pdf} = f(x; \gamma, \alpha, \beta) = \begin{cases} 0, & x<\gamma \\ [\alpha/(\beta^{\alpha})](x-\gamma)^{\alpha-1} \exp[-[(x-\gamma)/\beta]^{\alpha}], & x\geq\gamma \end{cases}$$

[Eq. 2]

$$\text{cdf: } F(x; \gamma, \alpha, \beta) = \begin{cases} 0, & x<\gamma \\ 1 - \exp[-[(x-\gamma)/\beta]^{\alpha}], & x\geq\gamma \end{cases}$$

[Eq. 3]

where:

a) x is the random variable under study. In our case, x can be interpreted as development time.

b) α is a shape parameter. It determines the width of the peak of the distribution and the expected error. We can associate this behavior with the efficiency of the project, which depends on characteristics of the process and the resources.

c) β is a scale parameter that stretches or compresses the graph in the x direction and hence controls the thickness of the tail. This parameter models the extra work introduced by new requirements or changes in requirements.

d) Note that the functions start at $x = 0$. We require a third parameter to shift the curves to the right. For that reason we introduce a location parameter γ, which is function of the already discovered system complexity.

CALIBRATION OF PARAMETERS

To calibrate productivity (α) and requirement's volatility (β), we conducted simulations with VitéProject [8, 13] using the following scenarios (fig. 4). Each scenario name consists of three letters describing the value for each of the three variables under study: productivity (α), requirements' volatility (β), and complexity (γ). Each letter could have two values: high (H) or low (L). The tool was configured to run 100 simulations for each scenario, and the organizational parameters were set to match the characteristics of software development.

Scenario	Productivity	Req. volatility	Complexity
LLL	Low	Low	Low
LLH	Low	Low	High
LHL	Low	High	Low
LHH	Low	High	High
HLL	High	Low	Low
HLH	High	Low	High
HHL	High	High	Low
HHH	High	High	High

Figure 4: Scenario's characteristics

To analyze the effect of productivity, we compared the results of the simulations of the following scenarios: LLL vs HLL, LLH vs HLH, LHL vs HHL, and LHH vs HHH. We found that for high productivity scenarios (Hxx) the development time improved by 60%.

To analyze the effect of requirement volatility, we compared the results of the simulations of the following scenarios: LLL vs LHL, LLH vs LHH, HLL vs HHL, and HLH vs HHH. We found that high requirement volatility (xHx) degraded the development time by 20%.

To analyze the effect of complexity, we compared the results of the simulations of the following scenarios: LLL vs LLH, LHL vs LHH, HLL vs HLH, and HHL vs HHH. We found that high complexity (xxH) degrade the development time by 30%.

6.1 Productivity (α)

Literature in productivity classifies time spent at work into four categories:

a) Direct. Time spent working and correcting errors on the product. In ViteProject terminology, it is the sum of work and rework.

b) Indirect. Time spent in activities supporting the work such as meetings, coordination, information exchanges, etc. In ViteProject terminology, it is known as coordination time.

c) Idle. Time spent without work to do, waiting for some input. In ViteProject terminology, it is known as waiting time.

d) Personal. Time spent doing anything except the other categories. ViteProject does not compute this category of time. However, it is loosely related to the noise parameter.

If we examine the time distribution of these categories we can observe a remarkable pattern that differentiates high productivity scenarios from the low productivity ones. This effect is independent of the other two variables of the simulation. Hence, this suggests that the time distribution can be a good indicator for the parameter α.

Figure 5 presents the distribution times for the eight scenarios simulated. A pattern of time distributions can be clearly observed. Scenarios with low productivity have a percentage of idle time greater than 13% of the total development time.

We can recognize low productivity scenarios also by the ratio of the percentage of direct time over percentage of idle time, which we call productive ratio (PR):

$$PR = \alpha = Direct\% / Idle\% \qquad [Eq. 5]$$

For high productivity scenarios $2.0 < PR < 6.0$, and for low productivity scenarios $0.8 < PR < 2.0$.

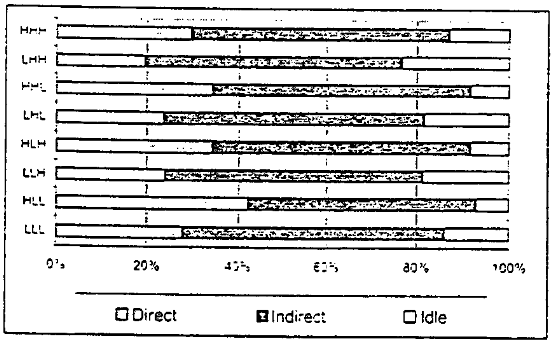


Figure 5: Time distribution from each scenario

We observed that using PR as the value of α, the model behaves as the simulations. That is on high productivity scenarios the total development is 60% shorter than in low productivity ones. The reasons why the ratio PR is related to productivity require further study. However, we conjecture the reason could be related to:

a) Fit of job and people skills.

b) People turnover, generating noise and productivity losses derived from training and learning curves.

c) Number of people, influencing the productivity by excess or default of working force.

In the model the use of α ranging from 0.8 (low productivity) to 6 (highest productivity), corresponds to the results observed in the simulations.

6.2 Requirement's volatility (β)

β, the extra delay factor caused by requirements' volatility (late requirements and changes in previous requirements), is obtained by the following formula:

$$\beta = \text{INT}((BR + DR) / 10) \qquad \text{[Eq. 6]}$$

Our simulations showed a 20% increase on the development time when the requirement's volatility is high.

6.3 Complexity (γ)

Having found a complexity metric suited for our purpose, the next step was to find for the existence of some sort of relationship between LGC and development time.

We conducted a simple experiment using the conversion ratio [Eq. 1] to obtain the size inputs for the sample. We used sample points from 1000 LGC to 30000 LGC, which means sample projects from 32 KLOC to almost 1MLOC. We compute the average estimation for the development time using COCOMO and Putnam. The sample points are plotted with a smoothing thick line. The logarithmic trendline is plotted as a thin red line. We found a strong logarithmic correlation ($R^2 = 0.9699$) with the following function (Fig. 6).

$$\text{Time (months)} = \gamma = 13\ \text{Ln}(LGC) - 82 \qquad \text{[Eq. 7]}$$

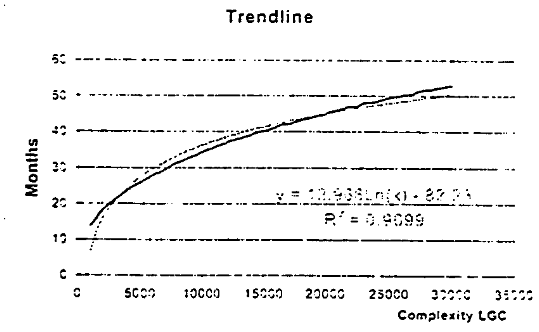


Figure 6: Complexity-time correlation

This equation gives a conservative estimation for projects between 4000 and 20000 LGC (128 and 640 KLOC of Ada). The estimation seems to be too optimistic for projects smaller than 2000 LGC or greater than 25000 LGC. Figure 9 shows the effects of complexity over different scenarios. The development time increases by 20% when the complexity is high.

6.4 The complete model

Our model requires three parameters (α, β, γ) that can be derived from metrics automatically collected from the development environment (Eq. 5, 6 and 7). If the development environment does not have the functionality to collect metrics, then a manual procedure could provide the data. Using these values in Eq.3 we obtain the probability of finishing the project at any given time (x in months) (Fig. 7). The model enables to refine the estimation form the knowledge captured at each evolutionary cycle. As the development progress γ increases (known complexity) and β decreases (less tail).

CONCLUSION

We introduced a formal method for risk assessment that solves the issue of human dependency, characteristic of the current risk assessment methodologies. This method is supported by a small set of metrics that can be automatically collected from the development environment.

One of the metrics introduced, productivity ratio, constitutes an objective method to assess the productivity level of an organization without subjective judgement of experts.

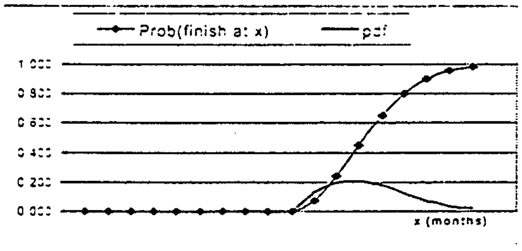


Figure 7: Distribution functions

We introduced a complexity metric well suited for real-time systems that has strong correlation with development time. Although, this metric was developed specifically for PSDL, the method can be generalized for other methodologies using Object Points or number of classes instead of LGC.

An interesting side effect of the model is that provides an easy way to estimate, very early in the life cycle, the duration of a project, and indirectly, its cost. This method enables an earlier assessment of the duration of the project and solves the problems of:

a) Human dependency on risk assessment, and
b) Difficulties in estimating time on evolutionary prototyping software processes.

Further research is required to generalize the method for larger systems and for different domains.

REFERENCES

1. Albrecht, A. Measuring Application Development Productivity. Proceedings IBM. October 1979.
2. Albrecht, A. and Gaffney, J. Software Function Source Lines of Code and Development Effort Prediction. IEEE Transactions Software Engineering, SE-9, 1983.
3. Boehm, B. A Spiral Model of Software Development and Enhancement. Computer. May, 1988.
4. Boehm, B. Software Engineering Economics. Prentice Hall, 1981.
5. Boehm, B., Madachy R., Selby, R. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. http://sunset.usc.edu/COCOMOII cocomo.html
6. Hall, E. Managing Risk. Methods for Software Systems Development. Addison Wesley, 1997.
7. Humphrey, W. Managing the Software Process.

Addison-Wesley. 1989.

8. Jin. Y. and Levitt. R. (Department of Civil Engineering. Stanford University). The Virtual Design Team. Paper to appear in Computational and Mathemetical Organization Theory. 1996.
9. Karolak. D. Software Engineering Management. IEEE Computer Society Press. 1996.
10. Kitchenham. B., Kansala. K. Inter-item Correlations among Function Points. First International Software metrics Symposium. IEEE Computer Society Press. 1993.
11. Kitchenham. B.. Linkman. S. Estimates. Uncertainty. and Risk. IEEE Software. May-June. 1997.
12. Kemerer. C. Reliability of Function Points Measurements: A Field Experiment. Communications of ACM. Vol 36 No 2. 1993.
13. The ViteProject Handbook: A User's Guide to Modelling and Analyzing Project Work Processes and Organizations. Vité C. 1999.
14. Londeix. B. Cost Estimation for Software Development. Addison-Wesley. 1987.
15. Luqi and Ketabchi. M. A Computer-Aided Prototyping System. IEEE Software. March. 1988.
16. Putnam. L. Software Cost Estimating and Life-cycle Control: Getting the Software Numbers. IEEE Computer Society Press. 1980.
17. Putnam. L. and Myers. W. Measures for Excellence. Reliable Software On Time Within Budget. Yourdon Press. 1992.
18. Putnam. L. and Myers. W. Executive Briefing. Controlling Software Development. IEEE Computer Society Press. 1996.
19. Putnam. L. and Myers. W. Industrial Strength Software. Effective Management Using Measurement. IEEE Computer Society Press. 1997.
20. Software Engineering Institute. Software Risk Management. Technical Report CMU/SEI-96-TR-012. June. 1996.

A Risk Assessment Model for Software Prototyping Projects[1]

Juan Carlos Nogueira
jcnoguei@nps.navy.mil

Luqi
luqi@cs.nps.navy.mil

Swapan Bhattacharya
swapan@cs.nps.navy.mil

Naval Postgraduate School
2. University Circle
Monterey, CA. 93943 USA

Abstract

Software prototyping processes have contributed to develop cheaper, faster and more reliable products. However, despite the advances in technology, little progress has been done in improving the management of software prototyping development projects. Research shows that 45 percent of all the causes for delayed software deliveries are related to organizational issues [1]. This paper addresses the risk assessment issue, introducing metrics and a model that can be integrated with prototyping development processes.

1. Introduction

Despite 50 years of progress, the software industry remains immature to meet the demands of an information-age economy. Many researches have treated the problem using different approaches: formal methods, prototyping, software processes, etc. However, this assertion remains true today. Experience suggests that building and integrating software by mechanically processable formal models leads to cheaper, faster and more reliable products [2]. Software development processes such the hypergraph model for software evolution [2], or the spiral model [3], have improved the state of the art. However, they have a common weakness: risk assessment. On the software evolution domain, risk assessment has not been addressed as part of the model. In the various enhancements and extensions, the graph model did not include risk assessment steps, hence risk management remains as a human-dependent activity that requires expertise. On the evaluation of the spiral model, one of the difficulties mentioned by Boehm was: *"Relying on risk-assessment expertise, the spiral model places a great deal of reliance on the ability of software developers to identify and manage sources of project risk."* [3].

Many researches have addressed the problem of risk assessment following the perspective of the traditional disciplines. The available tools for risk assessment are guidelines for practices, checklists, taxonomies of risk factors and few metrics. All these methods work fine if a) there is a human educated on risk assessment, and b) he/she has enough experience. Such resources are very scarce. Our research is focused on software project risk assessment, which in other words is the prediction of success of the project. The only way to evaluate the degree of success of a project is: a) to compare the planned and actual schedules; b) to compare the planned and actual costs; and c) to compare the planned and actual product characteristics. An emergent branch of software engineering has covered this last part: software reliability. However, we think that more emphasis put on in the first two. We believe that evolutionary prototyping provides the most promising context to address these issues.

1.1. Impact of evolutionary software processes

Studies have shown that early parts of the system development cycle such as requirements and design specifications are especially prone to errors [2]. Problems originating in the early stages often have a lasting influence on the reliability, safety and cost of the system. This effect is particularly notorious in projects involving multiple stakeholders with different points of view. Evolutionary prototyping offers an iterative approach to requirement engineering to alleviate the problems of uncertainty, ambiguity and inconsistency inherent in the process. Moreover, prototyp-

[1] This research was supported by the US Army Research Office under grant #38690-MA and grant #40473-MA.

0-7695-0668-2/00 $10.00 © 2000 IEEE

ing can improve the capture of change in requirements and assumptions during the development process.

Evolution-driven CASE tools for computer-aided prototyping provide logical assessment of the consistency and clarity of requirements and specifications. The use of prototypes facilitates the requirement phase in any type of software projects. Particularly, in real-time applications where severe time constraints impose more challenges, the use of prototypes facilitates to describe the requirements in a clear, precise, consistent and executable format. Prototypes are useful to demonstrate system scenarios to the affected parties as a way to: a) collect criticisms and feedback that are sources for new requirements; b) early detection of deviations from users' expectations; c) trace the evolution of the requirements; and d) improve the communication and integration of the users and the development personnel.

Despite the unquestionable benefits of prototyping, we have two concerns. First, the risk assessment issue has not been solved. The second concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required constructing the product. Most parts of project management and estimation techniques are based on linear layouts of activities, so they do not fit completely.

1.2. The estimation problem

In order to assess the risk in a project, it is necessary to have an idea of the effort and time involved. The industry has been using three classes of tools to estimate effort and time that can be applied at different moments during the life cycle, each category being more precise than the previous one but arriving later:

a) Very early estimations. This category includes very crude approximations done during the beginning of the process usually by subjective comparisons using previous projects.

b) Macro models. This category includes Basic COCOMO, Putnam, Function Points, etc. The estimation is done after completing the requirements phase.

c) Micro models. This category includes intermediate and detailed COCOMO, and Pert/CPM/Gantt techniques. The estimation is done after the design when it is possible to have a work breakdown structure. The project estimate is the integration of all module estimates.

It is not our intention to discuss these techniques, the details can be read in [4], [5], [6] and [7]. However we highlight the assumptions for COCOMO and Putnam's methods. COCOMO assumes:

(1) The development period starts at the beginning of the design phase. That means that the requirements phase is already done.

(2) The estimation covers only the direct-charged labor. In other words, time spent in meetings and communication is not considered.

(3) The model assumes that a rather optimistic working-time of 152 hours of productive work per month.

(4) The model assumes that the project will enjoy "good management."

(5) Finally, the model assumes that the requirements will remain unchanged. This is a really restrictive assumption that does not match the evolutionary prototyping process.

The other de facto standard, Putnam's model, is based on the following assumptions:

(1) A development project is a finite sequence of purposeful, temporally ordered activities, operating on a homogeneous set of problem elements, to meet a specified set of objectives.

(2) The number of problem elements is unknown but finite.

(3) Problems are detected, recognized and solved by applying effort.

(4) The occurrence of problem solving follows a Poisson process.

(5) The number of people working in the project is proportional to the number of problems ready to solve at that time.

(6) The requirements are done, which is very restrictive considering evolutionary software processes.

None of these techniques consider the following characteristics of software projects: a) requirement volatility, b) personnel volatility, and c) time consumed by communications, exceptions and noise in the process. All the methods use size as input parameter via some kind of derivation from complexity. In many cases the methods to compute such complexities and sizes are questionable. Recently, Stanford University [7] developed a new generation micro-model estimation tool (ViteProject) that addresses some of our concerns. This tool is useful but requires a complete work breakdown of the project, thus it is useful to control the project but cannot be used for early estimations. However, it is very useful to simulate different scenarios. We are using this approach to calibrate our model.

2. Metrics

Metrics is a key factor in the identification of threats. Without metrics it is not possible to provide early alerts of risks. In this section we describe a set of metrics that support our risk identification strategy. We decided to use a small set of metrics presenting the following characteristics: a) robustness, b) repeatability, c) simplicity in terms of the number of parameters, d) easy to calculate, and e) automatically collectable.

2.1. Metrics for Requirements

We define *birth rate* (BR) as the percentage of new requirements incorporated in each cycle of the evolution process. This metric shows the explosion of new requirements as a percentage.

BR = (NR / TR) * 100, where (Eq. 1)

NR = number of new requirements,

TR = total number of requirements (including NR).

We define *death rate* (DR) as the percentage of requirements that are dropped by the customer in each cycle of the evolution process.

DR = (DelR / TR) * 100, where (Eq. 2)

DelR = number of requirements deleted,

TR = total number of requirements (before deletion)

We define *change-rate* (CR) as the percentage of requirements changed from the previous version.

CR = (ModR / TR) * 100, where (Eq. 3)

ModR = number of requirements changed,

TR = total number of requirements.

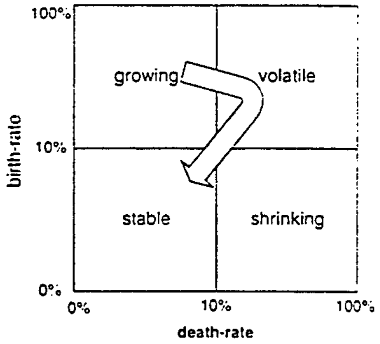


Figure 1: Evolution of requirements

From the point of view of the metrics, a change on a requirement can be viewed as a death of the old version and a birth of the new one. The simplification just described, enables to compare birth rate and death rate in a bi-dimensional plot that shows four regions: stability region, growing region, volatility region and shrinking region. Each of these regions has different risk connotations. There is a normal evolution of the project as the time goes by. During early stages, it is normal for projects being in the growing region. However, if the project continues in this region after many cycles, or return to this region after visiting other regions, then something wrong could happen. In the first case, the requirement engineering could not be efficient. The second case could show evidence of late discovery of some cluster of hidden requirements. After some cycles, the project should leave the volatile region. If the project evolves to the shrinking region, and the requirements engineering is working right, there is evidence that the customers are cutting down the project. This can be the indicator of a severe cut in the budget. Finally, any involution to a previous region should be considered as evidence of threats. In such cases a detailed analysis is required to assess the causes of the anomaly.

2.2. Metrics for Personnel

In order to measure personnel both quantitative and qualitative metrics are required. The skill match between person and job is required to estimate the speed in processing information and rate of exceptions. On the quantitative side we propose to measure the number of people and the turnover. This last one provides information about the expected productivity losses due to training, learning curves and communications.

2.3. Metrics for Complexity

Complexity has a direct impact on quality because the likelihood that a component fails is directly related to its complexity. The quality of the product can only be determined at the end of the process. Hence, it is important to measure the complexity as predictor. This particularly useful in real time systems, which present special difficulties in terms of requirement engineering. Some requirements are difficult for the user to provide and for the analysts difficult to determine. The best way to discover these hidden requirements is via prototyping. Computer Aided Prototyping System (CAPS) [2] is a CASE tool specially suited for this task. It has a graphical easy to understand interface and mapped to a specification language, which in turns generates Ada code.

The prototyping process consists of prototype construction and modification (evolution) based on evolving requirements and code generation. Both construction and modification are exploratory activities with a common target: to satisfy multiple users with different and often conflicting points of view. Requirement engineering is a consensus driven activity in which mechanisms for conflict resolution and traceability of requirement evolution represent critical success factors.

Formal specifications are suitable for being analyzed to compute their complexity. In the case of CAPS, the tool generates specifications in a structured language called Prototyping Specification Design Language (PSDL). PSDL code has the following tokens: types, operators, data streams and constraints. Types are declarations of abstract data types required for the system. Operators and data streams are the components of a dataflow graph. Finally, constraints represent the real-time constraints that the system must support.

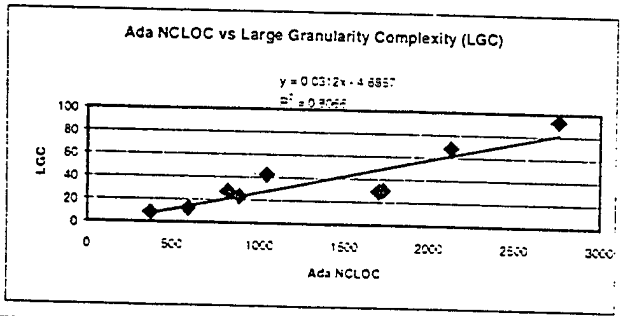


Figure 2: Correlation between non-comment Ada lines of code and LGC

We define two complexity metrics for PSDL: Fine Granularity Complexity metric (FGC), and Large Granularity Complexity metric (LGC). The reason to compute different metrics is because we want to detect two classes of threats. First, we need to be aware of operators that are too complex. High complexity on one operator could be caused by poor design and possible can be solved by further decomposition. Second, we require a metric to compute the total complexity of the system.

FGC expresses the complexity of each operator in the system and is a function of the fan-in and fan-out data streams related to the operator.

$$\text{FGC} = \text{fan-in} + \text{fan-out} \qquad \text{(Eq. 4)}$$

LGC expresses the complexity of the system as a function of the number of operators (O), data streams (D), and types (T).

$$\text{LGC} = \text{O} + \text{D} + \text{T} \qquad \text{(Eq. 5)}$$

We examined the correlation between LGC and size of the specifications and the code. We observed a very strong correlation between PSDL lines of code and LGC (R = 0.996). The correlation between Ada non-comment lines of code of the projects with their complexity measured using LGC, we observe a strong correlation also (R = 0.898) (Fig. 2). Even if CAPS generates part of the Ada code, the designer can add and modify the generated code introducing more variability. The following graph shows the correlation observed for the same set of projects. The size of the project in thousands of non-comment lines of code can be estimated as:

$$\text{KLOC} = (32\ \text{LGC} + 150) / 1000 \qquad \text{(Eq. 6)}$$

3. The proposed model

From the point of view of software engineering, it is necessary to create the methodology to solve the decision-making process during the early stages of the life cycle, when changes can be done with less impact on the budget and schedule. The most significant causes of software project failures are: lack of understanding of user's needs, ill defined scopes, poor management of project changes, changes in the chosen technology, changes in business needs, unrealistic deadlines, user's resistance, loss of sponsorship, lack of personnel skills, and poor management. From those pathologies, we conducted causal analysis arriving to the three risk factors that we will discuss.

We propose to divide risk management in three activities: risk identification, risk assessment and risk resolution. Risk identification is the set of techniques designed to alert and identity possible threats. Risk assessment is the quantitative analysis of the probabilities and impacts of the identified threats. Risk resolution is the application of resources and effort to avoid, transfer, prevent, mitigate or assume the risks.

In order to achieve risk management, an organization requires a minimum level of maturity that can be associated with CMM level 2 [8]. If an organization is not able to collect metrics, any attempt to formally identify and assess risks is impossible.

3.1. The risk major components

In our vision, software risks could be controlled if we could master how to administrate uncertainty, complexity and resources. Transforming the unstructured problem of risk assessment leads to a formal method able to be translated into an algorithm. In order to structure the problem, we proceeded to analyze the problem decomposing project risk into simpler parts. We used causal analysis to find the primitive threat factors. We identified three major factors: process risk, resource risk and product risk. Each of these factors introduces risks by themselves but mainly due to the interaction between them.

Resource risk, is affected by organizational, operational, managerial and contractual parameters such as resources, outsourcing, personnel, time and budget among others. The literature is abundant in this area. Various approaches use subjective techniques such as guidelines and checklists [9], [10], [11], which require expert's opinion even when they could be supported by metrics.

Engineering development work procedures such as software development, planning, quality assurance, and configuration management cause **process risk**. The more complex a process is, the more difficult it is to manage, and the more education, training, standards, reviews, and communication are required. Consequently, complexity grows. The software process complexity has been partially covered by research in terms of subjective assessments about maturity level and expertise [9], [10], [11]. However, we require a more precise and objective method.

Finally, **product risk** is related to the final characteristics of the product, its complexity, its conformance with specifications and requirements, its reliability and customer satisfaction. The product introduces its own risk factors in terms of quantitative and qualitative attributes. We identified two basic product-risk factors: requirement stability, and requirement complexity. Requirement stability is measurable using the set of metrics previously described. Due to lack of structure in informal requirements, it is necessary to transform them into specifications in order to compute complexity. Other product characteristics such as reliability and maintainability are not of interest to identify and assess risk on early stages. Reliability can be measured only after completion or almost completion. Maintainability can be measured only after the design is started. Both measures are useful to control the project in future phases. These estimations are useful in order to: a) identify the trade-off function between error reduction and cost of error reduction, b) provide quantitative basis for accepting or rejecting software during functional testing, and c) provide quantitative basis for deciding whether additional testing is warranted based on the cost of error removal.

The process provides the description of its environment and the theoretical requirements to execute it. Consequently, the process introduces threats due to its requirements and characteristics: complexity, technology required, budget required, schedule required, and personnel skills required. The resources represent the actual allowances in personnel, tools, budget and schedule. They impose constraints that could not match the process requirements. The productivity is consequence of the matching of these two facets of the project.

The decomposition created by causal analysis revealed: a) a method to identify risks by comparing the degree of mismatching between the product and process characteristics, against the resource constraints; and b) candidate indicators to be used in an estimation model.

3.2. The formulation

We can consider software projects as experiments where its cost and schedule are the output measures. We know that software projects tend to overrun costs and schedule (this fact has been proved by research and industry). There are two possible ways to interpret the result of the experiment. One hypothesis is that this behavior is abnormal, and a consequence of lack of process maturity (SEI/CMM approach). Another hypothesis is that this could be a "false-abnormal" behavior assumed abnormal as consequence of inappropriate measurements.

How do we create a macro model that considers the previous concerns and is able to be used during the evolutionary prototyping stages of the process? Our hypothesis is that a Weibull's family distribution can model each of the evolution cycles. Lets discuss the meaning of each of the variables in the function:

x is the random variable under study. In our case, x can be interpreted as development time.

α is a shape parameter.. It reduces the variability narrowing the shape of the pdf.

β is a scale parameter that stretches or compresses the graph in the x direction.

We require a third parameter (γ) to shift the curves to the right as consequence of system's conceptual complexity reflecting learning/training delays. The functions for the pdf and cdf are then respectively:

$$f(x;\gamma,\alpha,\beta) = \begin{cases} 0, & x < \gamma \\ (\alpha/\beta^{\alpha})\,(x-\gamma)^{\alpha-1}\exp[-[(x-\gamma)/\beta]^{\alpha}], & x \geq \gamma \end{cases} \qquad \text{(Eq. 7)}$$

$$F(x;\gamma,\alpha,\beta) = \begin{cases} 0, & x < \gamma \\ 1-\exp[-[(x-\gamma)/\beta]^{\alpha}], & x \geq \gamma \end{cases} \qquad \text{(Eq. 8)}$$

The development life cycle can be visualized a succession of prototyping developments with increasing functionality followed by a final optimization that produces the system. Each of these phases has the same activity pattern, so its reasonable to suppose that the delivery time for each one has a probability distribution from the same Weibull family but with different parameters.

During each prototyping cycle a certain number of problem events occur. A problem event is an effort-consuming situation that introduces a certain amount of functional complexity to be solved (caused by a new requirement, a change on a requirement, or as the consequence of rework), and a certain amount of information exchange.

We suppose that the occurrence of problem events in each cycle follows a Poisson distribution with different mean for each cycle. So, the entire development life cycle is a non-homogeneous Poisson process. We assumed this distribution because:

(a) There exists a certain rate of occurrence of events.

(b) The probability of more than one event occurring in a time interval depends on the length of the interval.

(c) The number of events during one time interval is independent of the number received prior this time interval.

4. Validation

Our model has been calibrated and validated in two ways: a) internal consistency proved by mathematics and statistics; and b) black box validation by comparing its outputs in duration and effort with other available models. Figure 3 shows a comparison of duration estimates using COCOMO. Putnam and this model. Our model gives a conservative estimation for projects between 4000 and 20000 LGC (128 and 640 KLOC of Ada). For the comparison. we converted from LGC to Ada lines of non-comment code using (Eq.6). and then we applied the obtained size to COCOMO and Putnam's model. The estimation seems to be too optimistic for projects smaller than 2000 LGC or greater than 25000 LGC.in month.

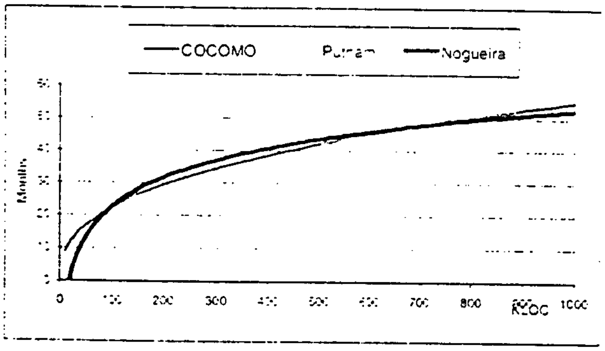


Figure 3: Comparison with COCOMO and Putnam methods

5. Conclusions

We addressed the issue of human dependency in risk assessment of the evolutionary software processes incorporating an automated risk assessment method integrated with evolutionary prototyping. Our approach provides a way to structure and automate the assessment of risk. The proposed model addresses part of the limitations of the traditional estimation methods. We are calibrating the model using simulations with ViteProject. Software development is still a human dependent activity requiring lots of human communication. and without appropriate managerial decision support tools. software engineering will remain in its present state. We think that we require improving our knowledge about the internal phenomenology of the software life cycle. It is in the human aspects of the software process where the bottleneck is located now. Automated risk assessment tools should consider these aspects. Without such knowledge, prototyping issues such as incomplete specifications. system complexity and development time will remain unpredictable.

References

[1] Van Genuchten. M. Why is Software Late? An Empirical IEEE Transactions on Software Engineering. June. 1991.

[2] Luqi and Goguen. J. Formal Methods: Promises and Problems. IEEE Software. January, 1997.

[3] Boehm. B. A Spiral Model of Software Development and Enhancement. Computer. May. 1988.

[4] Boehm. B. Software Engineering Economics. Prentice Hall. 1981.

[5] Putnam. L. and Myers, W. Industrial Strength Software. IEEE Computer Society Press. 1997.

[6] Londeix. B. Cost Estimation for Software Development. Addison-Wesley. 1987.

[7] The ViteProject Handbook. Vité ©. 1999.

[8] Carr. M. Risk Management may not be for everyone. IEEE Software. May - June 1997.

[9] Software Engineering Institute. Software Risk Management. CMU/SEI-96-TR-012. June. 1996.

[10] Hall. E. Managing Risk. Methods for Software Systems Development. Addison Wesley. 1997.

[11] Karolak. D. Software Engineering Management. IEEE Computer Society Press. 1996.

[12] Humphrey. W. Managing the Software Process. Addison-Wesley. 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52 2
Naval Postgraduate School
Monterey, CA 93943-5100

3. Research Office, Code 09 1
Naval Postgraduate School
Monterey, CA 93943-5000

4. Dr. David Hislop 1
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

5. Dr. Man-Tak Shing, CS/Sh 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

6. Dr. Valdis Berzins, CS/Be 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

7. Dr. Luqi, CS/Lq 7
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943